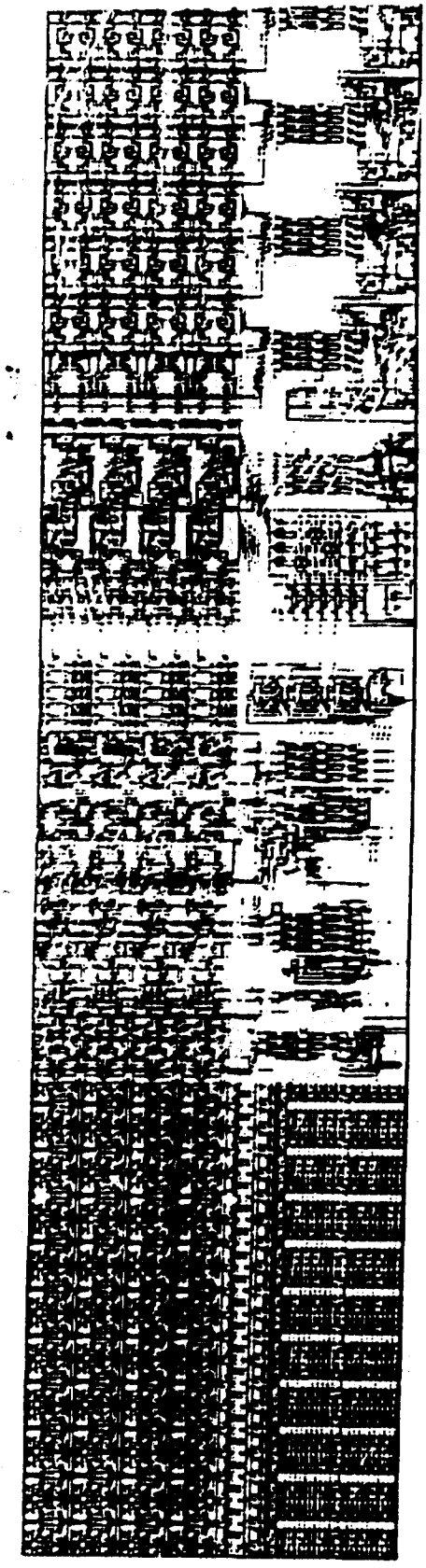


(2)

NORTHWEST LIS

(LABORATORY FOR INTEGRATED SYSTEMS)

Semiannual Technical Report No. 1
Dept. of Computer Science and Engineering
University of Washington

November 8, 1991
LIS TR #91-31-01

Reporting Period: July 1, 1991 – November 8, 1991

Principal Investigators: Gaetano Borriello
Carl Ebeling
Lawrence Snyder

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR #91-31-01	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Semi-Annual Technical Report #1	5. TYPE OF REPORT & PERIOD COVERED: Technical
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Gaetano Borriello Carl Ebeling Larry Snyder	8. CONTRACT OR GRANT NUMBER(s): N00014-91-J-4041
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Northwest Laboratory for Integrated Systems University of Washington Dept. of Comp. Science, FR-35 Seattle, WA 98195	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA-ISTO 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE: November 8, 1991 13. NUMBER OF PAGES: 40
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research - ONR Information Systems Program - Code 1513: CAF 800 North Quincy Street Arlington, VA 22217	15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Retiming, Field Programmable Gate Arrays, Subgraph Ismorphism, Timing Verification, High-Level Synthesis, Routing Networks, VLSI Testing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Northwest Laboratory for Integrated Systems
Department of Computer Science and Engineering
University of Washington

Semiannual Technical Report No. 1

November 8, 1991

LIS TR #91-31-01

Reporting Period: July 1, 1991 - November 8, 1991

Principal Investigators: Gaetano Borriello
Carl Ebeling
Larry Snyder

Sponsored by: Defense Advanced Research Projects Agency - ISTO
Issued by Office of Naval Research
Under Contract #N00014-91-J-4041

Technical Contact: Gaetano Borriello, 206/685-9432, fax: 206/543-2969
[gaetano@cs.washington.edu]

Administrative Contact: Kay Beck, 206/685-3796, fax: 206/543-2969
[kbeck@cs.washington.edu]

Table of Contents

1	Retiming of Level-Clocked Circuits	1
2	Triptych: A New Field-Programmable Gate Array Architecture	2
3	Subgraph Ismorphism	4
4	Symbolic Timing Verification and High-Level Synthesis	5
5	Synthesis of Microcontroller-Based Embedded Systems	7
6	Chaos Router	9
7	The MacTester	11

Appendices:

TRIPTYCH: A New FPGA Architecture
(Department of Computer Science and Engineering, University of Washington, TR #91-09-05)

OEsim: A Simulator for Timing Behavior
(Proceedings of the 28th ACM/IEEE Design Automation Conference)

Sizing Synchronization Queues: A Case Study in Higher Level Synthesis
(Proceedings of the 28th ACM/IEEE Design Automation Conference)

OPERATION/EVENT GRAPHS: A Design Representation for Timing Behavior
(Proceedings of the IFIP WG10.2 Tenth International Symposium)

1 Retiming of Level-Clocked Circuits

Carl Ebeling, Brian Lockyear

Using level-sensitive latches instead of edge-triggered registers for storage elements in a synchronous system can lead to faster and less expensive circuit implementations. This advantage derives from an increased flexibility in scheduling the computations performed by the circuit. In level-clocked circuits, a value may arrive early and flow through a latch, giving the following computation more time.

Taking full performance advantage of latches requires placing them in the circuit to achieve the best use of the clock cycle. This process of rearranging the storage elements in a circuit is called retiming and can be used to reduce the cycle time or the number of storage elements without changing the interface behavior of the circuit as viewed by an outside host. Retiming in effect reschedules the circuit computations in time based on the length of those computations. An efficient method for retiming circuits that use edge-triggered registers has been described by Leiserson, Rose and Saxe [LR83, LS91]. In essence, this method uses the clock period as the bound on the delay that can occur on a path in the circuit with no registers.

We have extended these retiming techniques to level-clocked circuits by first restricting the circuit domain to ``well-formed'' level-clocked circuits. In well-formed circuits, latches occur in order along any path through the circuit. This is the usual style of multi-phase circuit design and provides retiming with maximum flexibility for placing latches. We then define correctness of level-clocked circuits based on the proper flow of signals through the circuit. That is, we require signals departing one latch to arrive at the following latch during the next clock phase for that latch.

This definition of correctness leads to a set of simple path delay constraints and cycle delay constraints. The definition of a critical path between any two vertices is extended using these constraints, leading to constraints on the minimum number of latches on each critical path or cycle. These constraints can then be solved for any given clock period to find a valid retiming, if any. The usual search for the optimal clock period can then be performed. This technique is valid for clocks with any number of phases with no constraints on phase lengths or overlap other than the valid clock schedule constraints.

We are now working to relax some of the restrictions we have imposed, primarily the zero minimum delay constraint and the well-formed circuit constraint.

References:

[LR83] C. Leiserson and F. Rose and J. Saxe, "Optimizing Synchronous Circuitry by Retiming," in *Proceedings of the 3rd Caltech Conference on VLSI*, March, 1983.

[LS91] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, Vol. 6, No. 1, pp. 5-35, 1991.

2 Triptych: A New Field-Programmable Gate Array Architecture

Carl Ebeling, Gaetano Borriello, Scott Hauck, David Song, Elizabeth Walkup

Current general-purpose FPGAs use a combination of programmable logic blocks and programmable interconnect to provide a general circuit implementation structure. This clear separation between logic and interconnection resources is attractive because the mapping, placement and routing decisions are decoupled. The price for this separation is the large area and delay costs incurred for the flexible interconnection needed to support arbitrary routing requirements. This leads to architectures like Xilinx, where the routing resources consume more than 90% of the chip area.

Domain-specific FPGAs like the Algotronix CAL1024 and Concurrent Logic CFA6000 increase the chip area devoted to logic by providing a less general, nearest neighbor, routing structure appropriate for structured applications such as DSP and systolic algorithms. These FPGA architectures, however, are not suitable for general applications, particularly state machines and controllers.

We have designed a new FPGA architecture called Triptych which can efficiently implement both structured circuits like data paths and more general circuits like state machines. Triptych differs from other FPGAs by matching the structure of the logic array to that of the target circuits, rather than providing an array of logic cells embedded in a general routing structure. By matching the physical structure to the logical structure, we reduce the amount of random routing that is otherwise required. As shown in Figure 1, Triptych provides an underlying fanin/fanout tree structure that matches the general structure of multi-level logic DAGs.

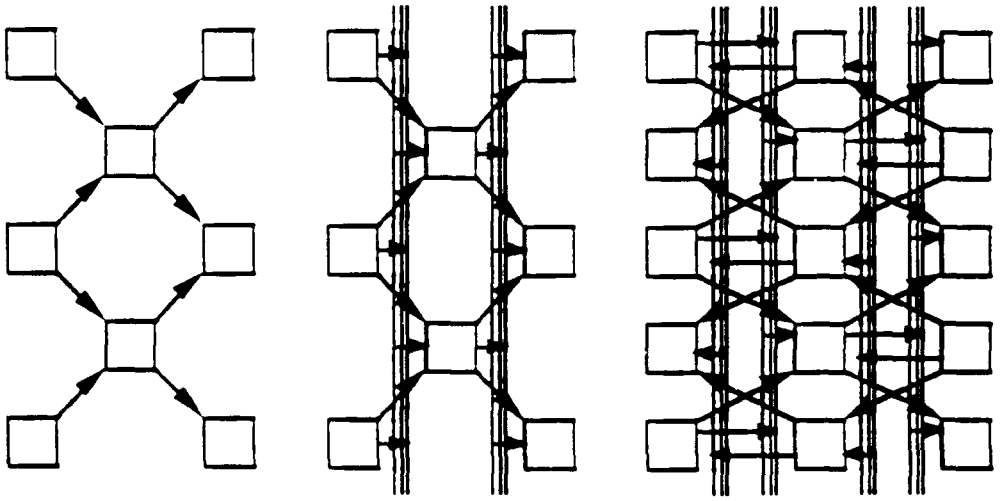


Figure 1. The overall structure of the Triptych FPGA shown in a progression of steps highlighting more and more features.

This basic structure is augmented with segmented routing channels between the columns that facilitate larger fanout structures than is possible in the basic array. Finally, two copies of the array, flowing in opposite directions, are overlaid. Connections between the planes exist at the crossover points of the short diagonal wires. It is clear that this array does not allow arbitrary point-to-point routing like that associated with Xilinx and Actel FPGAs. However, we claim that this array matches the form of a large class of circuits, and that a mapping strategy that takes this structure into account can produce efficient implementations.

We have measured the potential of the Triptych architecture relative to other reprogrammable FPGAs by manually mapping a range of interesting circuits, including structured circuits and random logic, and comparing the results with respect to area cost and circuit speed. This comparison shows that Triptych is similar in cost to the Algotronix CAL1024 for structured circuits suited to the CAL1024 and superior for circuits with non-local communication. Triptych is about twice as efficient as Xilinx across the whole range of circuits. The performance of Triptych implementations is comparable to that of other FPGAs.

Although Triptych shows considerable potential as an efficient FPGA architecture, the true viable of Triptych requires tools for automatically mapping circuits to its structure. We are now working on a set of mapping, placement and routing tools that incorporate information about the underlying routing structure and attempt to satisfy routing constraints early in the placement process. Our initial goal is to reach break-even point with respect to Xilinx which occurs at about 30% utilization. That is, if we can place and route circuits utilizing more than 30% of the available Triptych logic, then our area cost will be less than Xilinx. We believe that 50-75% utilization is ultimately realizable.

The complete text of a paper on Triptych that was present at the recent Oxford FPGA workshop appears as an appendix to this report.

3 Subgraph Isomorphism

Carl Ebeling

We have been working on a "daughter of Gemini" algorithm [EB83, EB88] for performing fast subgraph isomorphism for circuit graphs. Such an algorithm will be useful for automatically identifying components in large VLSI circuits so that hierarchy can be extracted from layout and perhaps in technology mapping for identifying possible coverings of logic graphs by library components.

Our algorithm is based on graph coloring and operates in two phases. In the first phase, the subgraph is colored such that the color of each node is determined only by the internal structure of the subgraph. A node at the center of the subgraph is also identified as the keystone node. The target graph is also colored such that the keystone nodes of all subgraph instances (and possibly other nodes as well) have the same color as the subgraph keystone node.

In the second phase of the algorithm, each possible keystone node is examined in turn to verify that it is part of a subgraph instance. This is performed by coloring from the keystone node in both the subgraph and the target graph. Fortunately, it is relatively easy to show that nodes outside the subgraph instance can be kept from causing spurious colors on internal nodes. This coloring provides a match if one exists, but must rely in difficult cases on backtracking.

A prototype implementation of this algorithm has been completed and preliminary results indicate that the algorithm works well for practical circuits. In the future we will be modifying some of the data structures to optimize the performance of the program, and measuring the performance for large graphs and difficult subcircuits.

References:

[EB83] C. Ebeling and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison," in *Proceedings of ICCAD*, pp. 172-173, 1983.

[EB88] C. Ebeling, "GeminiII: A Second Generation Layout Validation Program," in *Proceedings of ICCAD*, pp. 322-325, Nov. 1988.

4 Symbolic Timing Verification and High-Level Synthesis

Gaetano Borriello, Tod Amon

Symbolic timing verification is a powerful extension to traditional constraint checking that allows delays and constraints to be expressed as symbolic variables. The verifier then determines the relationships between these parameters based on known propagation delays and the timing constraints to be satisfied. This type of verification provides a way for synthesis tools to derive delay constraints on internal functions, given interface, throughput, and latency constraints provided by the user. We have developed an approach to symbolic timing verification using constraint logic programming techniques. The techniques are quite powerful in that they yield not only simple bounds on delays but also relate the delays in linear inequalities so that tradeoffs are apparent. We model circuits as communicating processes and our current implementation can verify a large class of mixed synchronous and asynchronous specifications.

Symbolic timing verification can be used to answer many other questions about a design specification and implementation. If circuit delays are known, then the verifier can still provide an answer as to whether or not a particular implementation meets all the timing constraints in the specification (as in current non-symbolic verifiers). More generally, however, using variables for delays can provide an answer about the range of values assignable to that variable that will still meet the constraints. This can be valuable information for a synthesis tool that must decide how to implement that particular function. At low-levels of granularity for circuit functions it may lead to a different logic network being used for a combinational function. At a high-level it may lead to a different architecture to implement a computation (e.g. more or less parallel). When many delays are represented by variables, the answer may be a set of linear inequalities constraining the variables. These relations provide synthesis tools with information about tradeoffs between circuit delays and how implementation choices for circuit functions may affect each other.

Therefore, a symbolic timing verifier is extremely valuable for synthesis. The information it produces about circuit delays can be used to determine how much time is available for a sequence of operations so that any available slack can be exploited in minimizing the circuitry required. As user-specified timing constraints change, the synthesis process may lead to very different circuit implementations that either exploit relaxed constraints to minimize area or use higher-performance architectures and components to meet tighter requirements. The utility of symbolic timing verification can be further extended if we consider symbolic timing constraints. In this case, the verification tool serves as an analysis tool that can determine how circuit delays relate to the symbolic constraints. For example, if we wish to determine the maximum throughput of a circuit, a variable can be used on the throughput constraint and the verifier can determine its range of values given circuit delays and other constraints.

It is these uses of symbolic timing verification, determining delay flexibility in synthesis and how it affects the resulting circuit architecture, that motivates this work. The information obtainable from a symbolic verification

process has three principal uses: (1) implementation verification, (2) obtaining constraints for synthesis, and (3) design evaluation. Implementation verification confirms that an implementation of a design and its associated delays will meet the constraints in the original specification. Synthesis tools can use symbolic delay values to determine the degree of flexibility that is available while still satisfying the timing constraints. This can lead to much more efficient use of resources in the final implementation. Design evaluation can be performed by using symbolic values in the constraints and determining bounds on the values of these variables given circuit delays. This can provide information about how well a design will perform and also relate the constraint variable to circuit delays.

5 Synthesis of Microcontroller-Based Embedded Systems

Gaetano Borriello, Pai Chou, Ross Ortega

Most digital design involves the design of controllers and most of the controllers currently on the market usually involve an embedded microcontroller that orchestrates the behavior of the system. However, high-level synthesis tools have yet to target this type of low-cost, yet ubiquitous, implementation medium and still focus on custom integrated circuit implementations. We have recently begun a project to address this required change in emphasis.

The issue is even more urgent as integration levels are increasing to the point where designers of these control systems have little idea what to do with all the devices that are now available on a single-die or within a single package. In response to this, highly programmable logic arrays are being marketed that can take advantage of economies of scale while still providing hardware speeds and effectively the same density at the board level as custom logic. Field-programmable gate arrays are not the only instance of this. Microcontrollers, with ever increasing options for their communication ports, are becoming extremely commonplace. There are even cases of the two being integrated onto the same chip, thus allowing an entire control system to be implemented in a single device that includes the required program and data memory.

Our project is to study all aspects of synthesizing to these types of devices. Problems range from how control-dominated designs are specified, to how tradeoffs can be made as to what functions will go in hardware and which in microcontroller software. Our approach is based on transformation of communication channels between the parallel processes that make up the system. We start from an initial specification and determine where a cut can be made between processes, separating them on the basis of hardware or software implementation. The position of the cut is based on the communication channels crossing it and whether they can be effectively and efficiently mapped to the microcontroller's I/O ports. Of course, transformations will be required to change the communication so that an optimized cut can be made. Criteria for guiding the transformation include width and speed of the communication channel as well as the size of the processes. Bandwidth and speed requirements have to be met. Program or hardware size may be under constraints related to the details of the microcontroller and programmable logic being used. Tradeoffs between the two media are necessary to get designs to fit.

Currently, we are in the initial phases of implementing our system. We are using a subset of Verilog with some specialized macros as our front-end and plan to generate Verilog output as well. The output processes will be tagged as being hardware or software and then sent to the appropriate low-level synthesis tools for final mapping. Most of our efforts are going into the development of a library of transformations for communication channels. Each transformation module will create new processes of either hardware or software to replace an existing communication channel and improve one of the above metrics. Also, we are investigating parallelization and sequentialization algorithms for communicating sequential processes, as these

will be needed to meet constraints or map multiple processes to a single microcontroller.

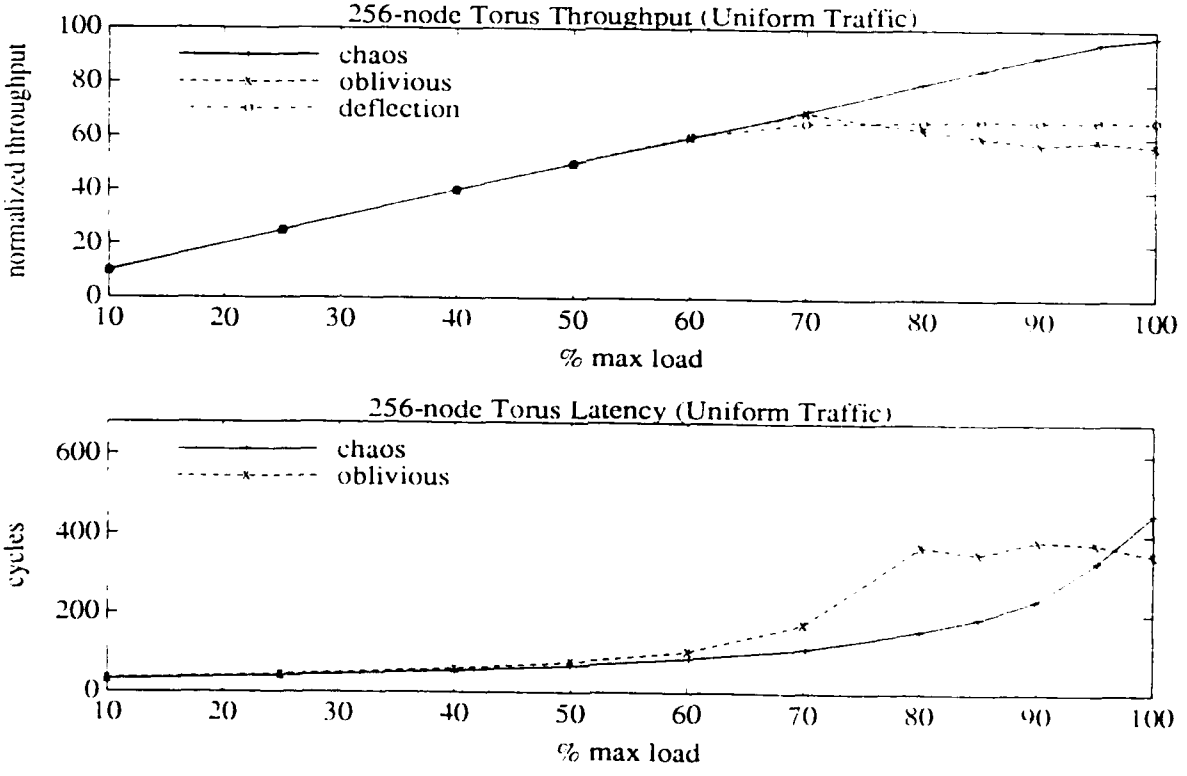
6 Chaos Router

Kevin Bolding, Samson Cheung, Carl Ebeling, Larry Snyder

The Chaos Router is an randomizing, nonminimal adaptive packet router for use in communication systems of parallel computers [KS90, KS91]. The router is deterministically deadlock free, probabilisitically livelock free and well suited for k-ary, d-cube topologies. The present research thrust is to compare chaotic routing with other routing strategies, e.g. oblivious routing and deflection routing, to understand the circumstances under which it is superior to competitors and to estimate quantitatively the amount of the improvement. Additionally, the fault tolerance implications of the router are being investigated. [This research is funded by NSF Grant MIP-9013274 and ONR Grant N00014-91-J-1007.]

Since the last semiannual report, considerable progress has been achieved in the areas of performance analysis and fault tolerance.

To evaluate the performance of chaotic routing against known routing methods, an enormous number of simulations have been performed for a variety of topologies (hypercube, mesh and torus), a variety of load characteristics (uniform random and 4X hot spots) and for a number of system sizes (64, 256 and 1024). The voluminous data generated by these simulation cannot be easily distilled into a single characterization. However, the accompanying graph gives a typical summary. Shown below are results for an oblivious router, a deflection (hot potato) router, and chaotic router.



The topology is a torus of 256 nodes. The plots show normalized throughput (bisection bandwidth = 100%) and latency (in flit times) as a function of presented load. The results show that chaotic routing with shared channels is able to transmit nearly all of the presented load and to have the minimum latency among the different routers throughout most of the range. Further results for 2-dimensional structures are presented in [BS91a].

In the fault tolerance arena, a protocol has been developed to implement system level fault tolerance. The protocol includes detection of lost or blocked packets, fault diagnosis (including discovery of inoperative channels and processors), fault recovery (including a limited amount of system reconfiguration), and system restart. Detection is especially interesting in the chaotic router because there is no "worst case time for delivery" of a packet, and thus it is not possible to use the usual timeout of acknowledgment failure to identify lost packets. Details of the protocol can be found in [BS91b].

References:

[KS90] Smaragda Konstantinidou and Lawrence Snyder, "The Chaos Router: A Practical Application of Randomization in Network Routing," in *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures*, ACM, pp. 21-30, 1990.

[KS91] Smaragda Konstantinidou and Lawrence Snyder, "Chaos Router: Architecture and Performance," in *Proceedings of the 18th International Symposium on Computer Architecture, IEEE*, pp 212-221, May 1991.

[BS91a] Kevin Bolding and Lawrence Snyder, "Performance Study of Two-Dimensional Chaotic Routers," Technical Report 91-04-04, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, April 1991.

[BS91b] Kevin Bolding and Lawrence Snyder, "Overview of Fault Handling for the Chaos Router," in *Proceedings of the 1991 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, November 1991.

7 The MacTester

Carl Ebeling, Neil MacKenzie, Larry McMurchie

The MacTester is a low-cost functional tester for both VLSI chips and board-level designs. The test head datapath uses Xilinx FPGAs and provides 128 test signals, each of which can be dynamically assigned as input or output. The test head is a ZIF socket that accommodates a variety of DIP's and PGA's up to 132 pins without any auxiliary wiring, with the possible exception of power and ground for large chips. The MacTester was originally designed with an interface to the Mac Nubus. During the summer of '91, we completed the design of an interface to the IBM AT bus.

There are two software environments for running the MacTester. One consists of writing a test program (using any of ANSII C compilers) and linking a library of low level tester routines that set and observe pins of the DUT. Another environment is the DesignWorks schematic capture and simulation system (from Capilano Computing), where the DUT is represented by an icon just like any other schematic device. In this way one can add circuitry to the schematic drawing that sets values on the DUT input pins and observes/checks the output pins.

Recently, Applied Precision Inc. of Mercer Island, WA has indicated they will manufacture and sell a tester based on the MacTester design. Availability is scheduled for early 1992.

Apple Computer provided funds and equipment for the design of the MacTester. Funding for the MacTester software development was provided by an Software Capitalization Grant (NSF Grant #MIP-9018224). Through MOSIS, DARPA has provided a means of fabricating tester PCBs.

TRIPTYCH: A New FPGA Architecture

Carl Ebeling, Gaetano Borriello,
Scott A. Hauck, David Song,
Elizabeth A. Walkup

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

Technical Report 91-09-05
September, 1991

This paper appears in "FPGAs", a book that contains the proceedings of the Oxford Workshop on Field Programmable Logic, September 1991.

TRIPTYCH: A New FPGA Architecture

**Carl Ebeling, Gaetano Borriello, Scott A. Hauck,
David Song, Elizabeth A. Walkup**

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Existing FPGA architectures can be classified along two dimensions: reprogrammable vs. one-time programmable and general-purpose vs. domain specific. The most challenging class of FPGA architectures to design is the reprogrammable, general-purpose FPGA, of which Xilinx is the most well-known example. In this paper we describe Triptych, a new FPGA architecture that addresses two problems of current reprogrammable FPGAs: the large delays incurred in composing large functions and the strict division between routing and logic resources. Our studies indicate that Triptych is more area-efficient than current architectures and has comparable delay characteristics for a large range of circuits that include both data-path elements and control logic.

INTRODUCTION

The most common approach to field-programmable gate array architectures is to dedicate a portion of the total chip area to logic functions and the remainder to interconnection resources. The logic functions may be fixed or programmable, while the routing is usually highly programmable to ensure that a large percentage of designs are routable. The flexibility of the interconnection network is limited by two factors: the number of configuration points (bits or fuses) that can be accommodated on chip and the speed requirements of the signals routed through the network (more switches or fuses on a signal path imply slower wires) (Rose 1991).

FPGAs can be programmed using a reprogrammable memory-based scheme or a one-time programmable fuse technology. Xilinx is the most well-known example of a reprogrammable FPGA (Carter 1986). It has logic blocks that can perform arbitrary functions of five inputs. The routing resources are arranged in an orthogonal grid around the function blocks and occupy approximately 90% of the chip area. Approximately 300 function blocks can be placed in a single device, the number being limited by the extra routing resources additional function

cells would require. In a chip with 320 cells, 64,160 programming bits are required, or approximately 200 bits per cell (Xilinx 1991).

Among one-time programmable FPGAs, Actel is the most common (El-Ayat 1989). Actel arranges a basic cell in rows similar to an arrangement of standard cells in a semi-custom integrated circuit. The cell functionality is fixed, with the logic function determined by where inputs are connected to the cell (typical usage is as a 3-input function). The interconnection resources are also similar to the standard cell style with wires running in segmented channels between the rows of cells and orthogonally across the cells to provide routing in all four directions. The logic cells account for 10-15% of the chip area and 750,000 bits are required to program a typical chip of 1200 cells (Actel 1991). The number of routing tracks limits the total number of cells that can be placed with reasonable routability on a single chip.

A more recent entry in the FPGA arena is the Apple Labyrinth architecture (Furtek 1990). Rather than dedicating chip area to either computation or interconnect, the Labyrinth FPGA tiles the chip with identical small cells that can perform either 2-input functions or routing, depending on the user-specified programming in the 4 bits per cell. Each cell is connected only to its four nearest neighbors. The design is intended for pipelined bit-serial applications, because the delays incurred in routing through many cells severely limit the cycle time.

In this paper, we present an alternative structure for reprogrammable FPGAs that blends logic and routing resources more closely than most other FPGAs. That is, each routing and logic block (RLB) in the Triptych array can be used both to compute a logic function and route signals. More importantly, the array is structured to match the inherent fanin/fanout tree structure of circuit graphs. This allows the physical layout of a mapped circuit to follow its logical structure, reducing the need for extensive routing resources. Circuits use varying numbers of RLBs for routing depending on how much their structure diverges from the Triptych structure.

We decided to undertake the detailed design and implementation of Triptych in the graduate VLSI implementation course (CSE568) in the winter quarter of 1991. The problem was an ideal class project because there was only a small collection of basic cells to design, and students could work on implementation and mapping issues in parallel. This paper describes the basic Triptych architecture and the experience we gained implementing it and mapping circuits to it. The two sections following this introduction describe the architecture in detail and the issues and design choices encountered during implementation. The next section provides a first look at how the architecture can be used and how it compares to others, as well as some ideas for automatic mapping. Finally, we conclude with remarks about both the architecture and the educational experience.

TRIPTYCH

The FPGA architecture we present in this paper differs from other FPGAs by matching the structure of the logic array to that of the target circuits, rather than providing an array of logic cells embedded in a general routing structure. By matching the physical structure to the logical structure, we reduce the amount of "random" routing that is otherwise required.

Figure 1 shows a high-level view of a typical multi-level combinational logic circuit. The flow is shown as unidirectional, from inputs to outputs. From the point of view of each input, the data flow forms a fanout tree (shown with solid arrows) to those outputs that the input affects. From the point of view of each output, the data flow forms a fanin tree (shown with

dashed arrows) from those inputs it depends upon. It is this fanin/fanout tree form that Triptych emulates architecturally by arranging RLBs into columns, with each RLB having a short, hard-wired connection to its nearest neighbors in adjacent columns (see Figure 2).

The basic structure is augmented with segmented routing channels between the columns that facilitate larger fanout structures than is possible in the basic array. Finally, two copies of the array, flowing in opposite directions, are overlaid. Connections between the planes exist at the crossover points of the short diagonal wires. It is clear that this array does not allow arbitrary point-to-point routing like that associated with Xilinx and Actel FPGAs. However, we claim that this array matches the form of a large class of circuits and that mapping will produce routable implementations.

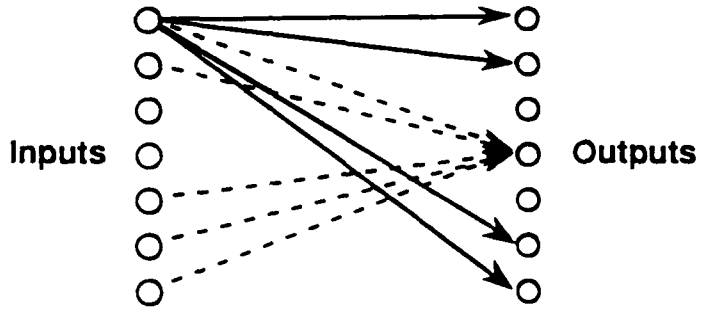


Figure 1 View of a multi-level combinational logic circuit as interleaved fanin/fanout trees.

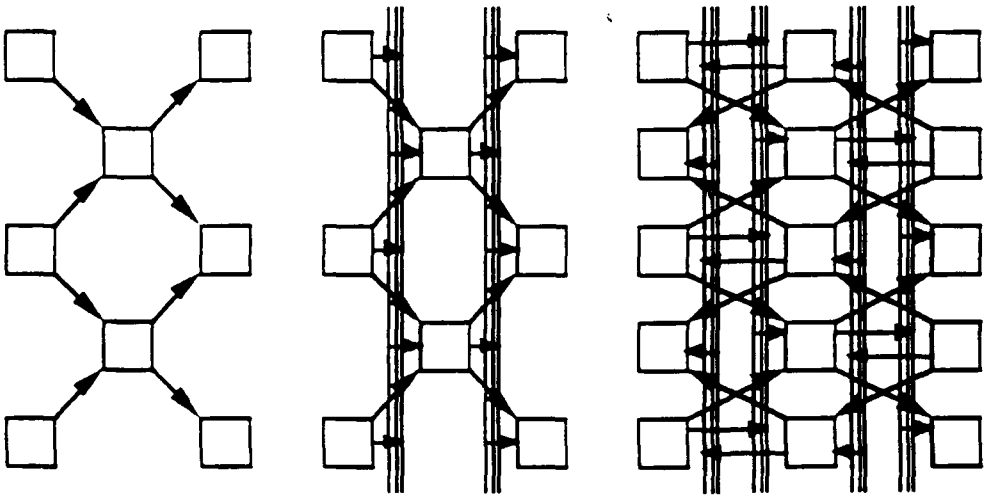


Figure 2 The overall structure of the Triptych FPGA shown in a progression of steps highlighting more and more features. The basic fanin/fanout structure on the left is augmented with segmented routing channels that make a third input and a third output available to the RLBs. The structure on the right is obtained by merging two copies of the middle structure, with data flowing in opposite directions in the two copies. Not shown are the connections between the two copies, which permit internal feedback.

Each RLB in the array has three inputs and three outputs and may perform an arbitrary logic function of the three inputs, with the result optionally held by a master/slave D-latch (Rose 1990). Routing in the Triptych array is in three forms: horizontally through the RLBs (by selecting an input to be routed to an output), diagonally through short wires to neighbors, and vertically through the segmented channels between columns of RLBs. Only one input and one output can be connected to the vertical wires; the other two must be on the local diagonal interconnect.

Circuits can be mapped onto this array by partitioning the logic into circuit DAGs containing nodes with at most three inputs. These DAGs are then mapped to the physical structure, with the inputs at one side of this structure and the outputs generated at the other. The nodes of the DAGs are placed such that input signals are available from the neighbor nodes or along a vertical connection. As Rose suggests in (Singh *et al* 1990), delay can be minimized by using mostly direct, hard-wired connections for the critical path. Triptych implementations do not strive for 100% logic utilization. Many RLBs will be used to provide routing, either to fanout a signal or to pass it forward to the next level. Sometimes a mapping will leave some cells unused to achieve a routable placement of nodes. Examples are provided below.

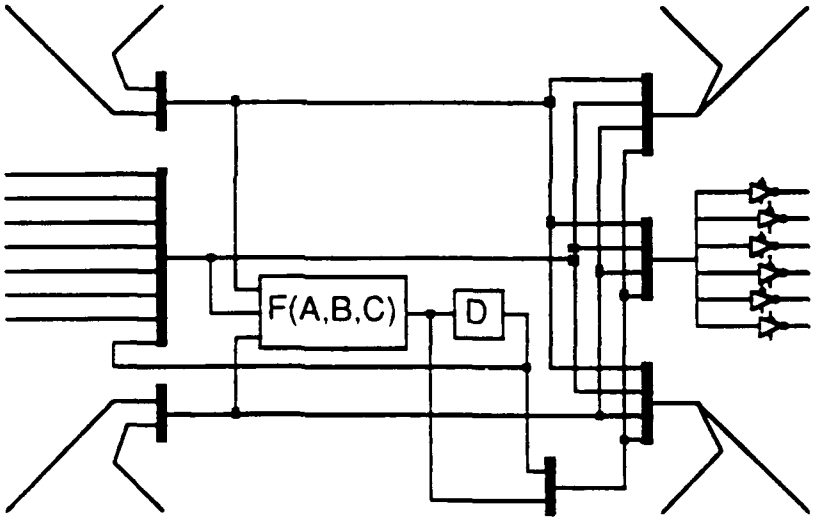


Figure 3. Triptych RLB design. The RLB consists of: 3 multiplexers for the inputs, a 3-input function block, a master/slave D-latch, a selector for the latched or unlatched result of the function, and 3 multiplexers for the outputs.

RLB structure

A logical schematic of the basic Triptych RLB is show in Figure 3. As can be seen, the cell is designed to handle both function calculation and signal routing simultaneously (hence the name *routing and logic block*, RLB). It takes input from three sources and feeds them into a function block capable of computing any function of the three inputs, and the output can then be used in latched or unlatched form. The RLB's three outputs can choose from any of the three inputs and either the latched or unlatched version of the function block output. One last feature is the

loopback from the master/slave D-latch, which enables the function to be dependent on its previous value. This last feature is included for state machine implementation, although it may be used to output both the latched and unlatched versions of the function block. Again, only one of the inputs and one of the outputs can be connected to the vertical wires: the other two of each type are connected to the local diagonal wires.

Typical RLB utilization

A Triptych RLB is capable of performing both function calculation and routing tasks simultaneously, which leads to several different uses of the RLB (see Figure 4). The three most obvious are: (a) a routing block with each input connected to one of the outputs; (b) a splitter with one of the inputs going to two or three of the outputs; and (c) as a function calculator with the three inputs going to the function block and the function going out the outputs. However, there are two important classes of hybrids that help produce more compact designs. The first comes from the observation that in blocks used to calculate a three-input function, the function block will most likely not go out all three outputs, and one or two of the input signals could be sent out the unused output connection(s), as in (d). Secondly, a function of two inputs can be implemented by making the function insensitive to the third input, thus allowing the unused input to be used to route an arbitrary signal, as in (e). An important observation is that the RLBs will never need to be used for one-input functions (i.e., an inverter), since any output signal will only be used either as an input to another arbitrary function block where the inverter could be just merged into the function computed, or to an output pin where an optional inversion can be applied.

As was shown earlier, the Triptych FPGA has no generalized interconnect for moving signals horizontally. Instead, there is a heavy reliance on unused RLBs and unused portions of RLBs to perform these routing tasks.

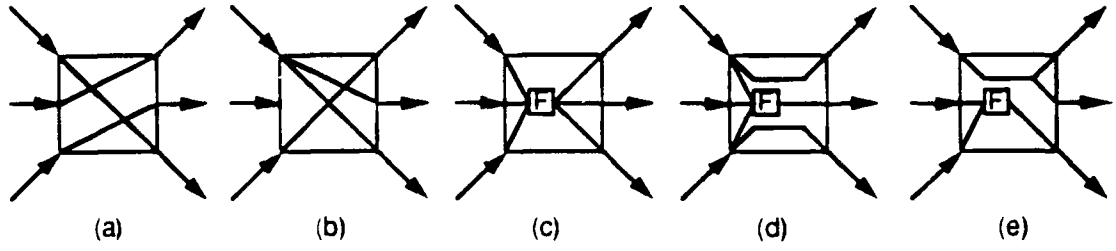


Figure 4 Five typical uses of Triptych RLBs.

Interconnection

The Triptych RLBs are connected by three separate interconnection schemes. The first is for horizontal interconnect and is accomplished through the RLBs as described above. The second is for local high-speed communication between neighboring RLBs and is achieved through "diagonals". The detailed structure of the diagonals is shown in Figure 5. They allow RLBs to send outputs to the RLBs immediately above and below them, which flow in the opposite direction, and to the two RLBs in the same position in the next column, which flow in the same

direction. Diagonals are important for two reasons. Diagonals permit the construction of multilevel functions of more than three inputs without the speed penalty of general-purpose interconnect. They also allow signal flow to change direction both so that circuits can be more tightly packed and feedback can be provided for the implementation of sequential logic.

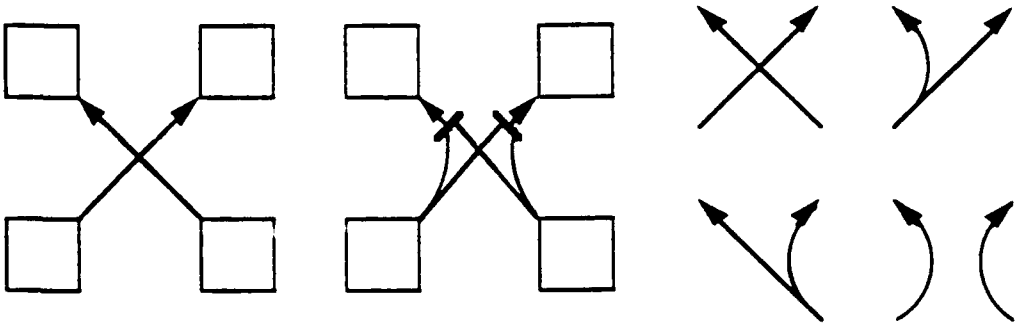


Figure 5 Schematic view of a pair of diagonals and the routing combinations they allow (implemented by a multiplexer at each diagonal input). The diagonals connect an RLB's outputs to the RLB's four nearest neighbors: two directly above and below in the same column and the two in the same positions in the next column.

The third type of interconnect is used for longer range connections and large fanout nodes. It is implemented as a set of segmented "channel wires" between adjacent columns (see Figure 6) that connect middle outputs of RLBs to the middle inputs of RLBs flowing in the same direction in the next column. Needless to say, this flexibility leads to a slower path, and speed-critical designs will avoid using the vertical channels for critical paths. There are 7 tracks in a vertical channel, with 6 handling inter-cell RLB routing and a seventh to carry a pin input. The 6 inter-cell tracks are broken up into two tracks each of 8, 16, and 32 RLB high segments.

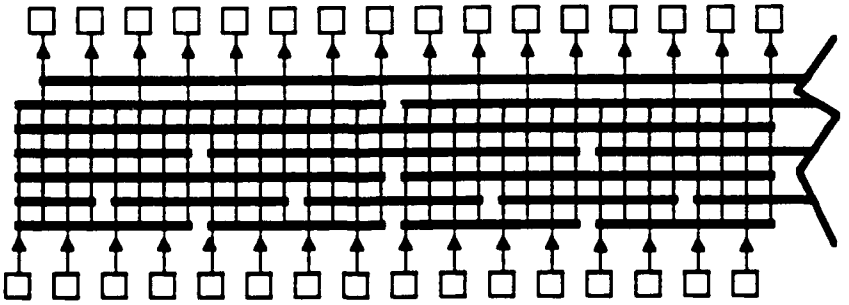


Figure 6 .Top half of a segmented channel (on its side). The bottom half is a mirror image of the top.

One last important feature of the interconnect structure is how it handles the array borders. Since there are no RLBs beyond the right and left edges for the channel wires to route to, the channels on the edges tie the two directions of RLBs together. This way of handling the border cases leads to a different way of looking at the array, namely as a cylinder of RLBs. If the diagonals leading to the opposite direction of RLBs were cut except for those at the edges, the chip would appear to be a folded cylinder of RLBs. In fact, it is often helpful to think of the array as containing many smaller cylinders. For example, a six by six square of RLBs can be broken off from the rest of the array and considered to be a cylinder three RLBs high and twelve RLBs in circumference. This is not quite true because the vertical channel for the left and right edges of the original six by six square will be unusable on the cylinder, however it can be a useful abstraction for hand mapping. In fact, the Triptych chip is an array of 64x8 RLBs, yielding a cylinder of 32x16.

Programming bit implementation and the scan path

Triptych is a RAM-based reprogrammable gate array with 26 memory bits per RLB, including those bits used for all three types of routing. The memory cells are implemented pseudo-statically with a "hold" signal asserted during normal operation and unasserted during programming. We found that this gave a much smaller layout than a fully static design (including the space needed for this extra hold line), especially when it was realized that the hold signal was necessary for selectively disabling RLB output drivers during programming. The memory cells are connected by a scan path running throughout the chip, allowing it to be programmed by cycling data through the bits.

The scan path used for programming is also attached to the RLB's master/slave D-latches. This not only allows the chip to start in any arbitrary combination of latch states, but it also allows the contents of the latches to be shifted out after the chip has run an arbitrary number of cycles to facilitate debugging. Also, if the scan path input is connected to the output, a programmed circuit can be stopped at any point, the contents of the D-latches analyzed, and the circuit resumed at the previous starting point.

Vital statistics

The speed of a path in a Triptych RLB can be calculated from the numbers given below in Table 1. For example, a path using 4 RLBs, 2 for routing and 2 for function calculation, and 1 channel wire would take 13.9±0.6 nanoseconds (4¥1.6 + 2¥2.2 + 3.1±0.6 = 13.9±0.6). Note that being able to use such a simple speed calculation method is due both to the simplicity of the interconnect and also to the design philosophy of "independence of paths" described below.

Table 1 Speed of important features, estimated using HSPICE with parameters for the 1.2mm CMOS n-well process available from MOSIS.

Resource Used	Delay
RLB	1.6ns
Function Block	additional 2.2ns

Channel Wire	2.5-3.7ns

Table 2 Estimated space and memory utilization per RLB of various features. (Note: percentage of RLB area includes area for memory cells.)

	Percentage of RLB area	Number of bits
Vertical Segmented Channels	54%	9
Diagonals	6%	2
Internal Routing & Multiplexers	23%	7
Function Block & D-Latch	17%	8
Total		26

Table 2 describes the relative sizes of the main components of the Triptych RLBs. The features measured are "Vertical Segmented Channels", including the line drivers and line readers; "Diagonals" which includes similar features as the "Vertical Segmented Channels"; "Internal Routing & Multiplexers" which includes the three 4:1 multiplexers for selecting the signal to send to each output as well as the 2:1 multiplexer that chooses between the latched and unlatched function block output; and "Function Block & D-Latch". Note that each category not only includes the area needed for the given functionality, but also the area necessary to store the configuration bits (which contributes 1% of RLB area per bit, since 26 programming bits take up 26% of the RLB area).

Probably the most important observation to be made from the table is that 83% of RLB area is devoted to routing of one form or another, with the actual function calculation only occupying 17%. Note that this number is fairly small compared to other reprogrammable FPGAs since a full 30% of the space for "Vertical Segmented Channels" is actually the inverters and tri-state buffers used to drive the channel wires, with another 6% in associated memory cells. These features would be included in the function blocks of other FPGAs.

DESIGN ISSUES

The design and implementation of the Triptych FPGA brought up several issues that we feel are of general interest. These are discussed in the following subsections.

Regularity

A goal in the design of the Triptych cell and interconnect was to achieve as regular a structure as possible. This was done because technology mapping is difficult, and any irregularities only complicate the issue. For example, the Triptych function block can compute any function of three inputs, as opposed to designs such as the Actel FPGA where only a subset of the possible functions can actually be realized in a cell. Also, an arbitrary function block removes the worry of what to do for inversions, since an inverter can easy be factored into any or all of the inputs and the output of the function block.

The interconnect scheme follows the same philosophy; the only deviation is caused by the edges of the array. A more creative structure with the interconnect optimized differently (e.g., as a butterfly) could have been implemented, but we feel that the complications added to the technology mapping stage would negate any potential gains.

Independence of paths and logical effort

The Triptych RLB is mostly composed of multiplexers and bus drivers. Early on, the decision had to be made whether to implement most of the multiplexers with switch or gate logic. Our original choice was to do most of the RLB in switch logic and only insert inverters where necessary to drive loads. We have since decided this was a mistake and have redesigned the circuit almost completely in gate logic. The main reason for this is something we call "independence of paths". The idea is that the routing of two different paths should affect the timing of each other as little as possible. This point is much the same as the one above, except that where the above rule dealt with the logical specification of the RLB and the interconnect, this deals with how the RLBs are actually implemented. Take for example the case where a single RLB output fans out to several inputs. If the RLBs were implemented in switch logic, with pass-gates taking inputs off the vertical channels, a signal would propagate more slowly if several RLBs were reading the same interconnect line than if each had its own. Thus, a technology mapper designed to optimize for speed would have to make sure that the critical path always used its own interconnect line. There are several other places where this effect can manifest itself, such as routing an input to an output (which slows down the function calculation) and splitting a signal to two or more outputs (which slows down both signals). This rule exists not just to make technology mapping easier; by making paths independent, it is also much easier to optimize the RLB channel wire drivers.

The Triptych chip was originally laid out by a handful of graduate students with little or no previous integrated circuit design experience. The project was carried on by one of these graduate students (Scott Hauck), who did a completely new layout aided significantly by the model of logical effort (Sutherland and Sproull 1991) which assists in the proper sizing of transistors and insertion of buffers to optimize speed. Although we have no firm numbers determining how much better the second design is than the first, we feel that logical effort can help novice designers develop faster circuits.

Routing flexibility

There are several unsettled issues in the design of the Triptych routing network. First and foremost is the sharing of tracks in the vertical segmented channels. By sharing tracks between RLBs flowing in opposite directions, we could implement a more flexible feedback capability than is possible using only the diagonals. Currently, the array has seven tracks for each direction, for a total of 14 in each segmented channel. One alternative is to have 5 tracks for each direction with another 2 shared for a total of 12. It is difficult to tell just how much sharing is needed. The shared tracks would have more drivers and receivers than they would if they were not shared and thus be slower. More experience with manual and automatic mapping will be needed before this issue is resolved.

Another issue relates to the D-latch loopback capability, which replaces the channel wire input in RLBs that use the loopback. Most likely, this input will be needed for an input and conflict with the use of the loopback. The loopback exists because it was extremely cheap to

include. The alternative is to route the output of the D-latch around through the RLB above or below on diagonals. Whether this is sufficient or a form of internal loopback is required (possibly coming in on diagonal inputs) will also be determined by experimentation.

Finally, we still must resolve how the Triptych array will be connected to the chip's I/O pins. Inputs can reach the array by entering RLBs on either vertical edge or by entering the vertical channels from the top and bottom. We expect to provide general input/output pads on all four sides with routing channels along the top and bottom of the array. Connections with either vertical edge will be more direct to provide a fast path into and out of the array for data-path applications.

Programming hazards

In FPGA design it is very tempting to ignore the programming phase, except to demand the most compact implementation of the programming bits as possible. However, this can lead to some serious problems. In an FPGA, there are certain assumptions made about the programming that are not enforced by the hardware. For example, it is assumed that at most one RLB drives any specific channel wire. This is automatically enforced in intra-cell routing and diagonals by virtue of multiplexer logic. In the case of channel wires, special hardware is required to enforce this constraint, with a very high overhead. In the Triptych FPGA, we simply assume that the software performing the mapping deals with this problem and that no configuration loaded will violate this constraint. During programming, however, bits stream through the programming memory, violating this programming assumption. This leads to short-circuits in the chip and possible damage. One solution is to adopt a bit-addressable scheme for the programming memory rather than a scan-path, but this is quite expensive due to the extra routing and decode logic required. Instead, we use the same signal that enables the scan-path to disable all channel wire drivers. Thus, while the chip is being programmed no drivers are active, thereby eliminating the problem. This costs approximately an extra 3% in chip area for the transistors and wiring required.

USING TRIPTYCH

In this section, we present several circuits that we have mapped by hand to Triptych. The purpose of these examples is to demonstrate the constraints on routing and how multilevel logic circuits do indeed map to the physical structure provided by Triptych. In these examples, each RLB is shown as a cell with three input entries and three output entries. Each entry indicates an incoming or outgoing signal. Note that each block may create a new signal by computing a logic function over the inputs. Diagonals and reverse diagonals that are used in the implementation are highlighted, as are connections to the channel wires. For clarity, only those vertical wires carrying signals are shown.

8-bit rotate function

The power of using columns of RLBs for routing only is shown in this example which rotates a set of 8 bits 4 positions. Each level can be used to send one signal from each RLB to a neighbor of the final position. Since each RLB has two outputs, one intermediate RLB column and two vertical channels are required to route the signals to their final destination (see Figure

7). This generalizes to the case where three signals are routed per RLB, which requires two intermediate RLB columns and three channels.

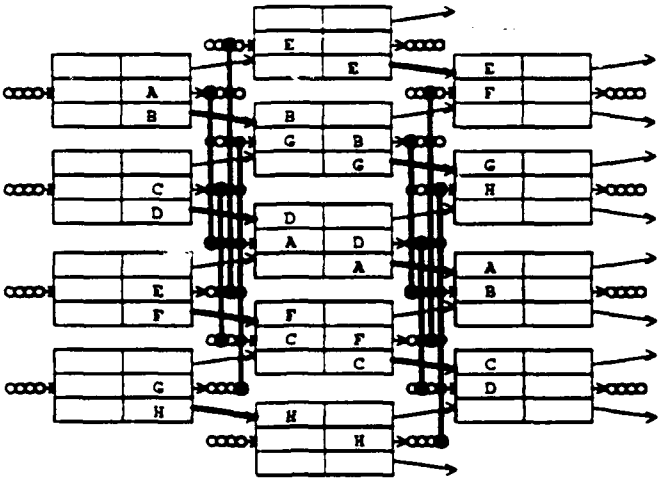


Figure 7 Triptych mapping of a 4-bit rotate of 8 bits.

Generalizing this use of the vertical channels suggests a naive place and route algorithm that alternates columns of RLBs used for routing with columns used to compute logic functions. Subject to a sufficient number of routing tracks, this leads to a viable routing of arbitrary logic functions. However, as the next example shows, this scheme is much less area-efficient than is generally achievable.

State machine example

Figure 8 shows the factored logic equations and corresponding Triptych implementation for the ubiquitous traffic light example. This example shows that circuit mappings can be very compact if the individual logic blocks are correctly placed. The inputs and outputs of this circuit are all connected at the left and right of the array, except for three signals that use the pin input track of the vertical channels (shown dangling off the bottom). In this example 16 RLBs are used to compute logic functions, 2 RLBs are used only for routing, and 6 RLBs are left unused (these 6 RLBs must be counted in order to achieve a rectangular mapping; they might be used in neighboring circuits). Also, this circuit is assumed to be placed along the left edge of the chip, so the vertical tracks at that edge are used to connect RLBs in the same column. Note that this example would have been easier to map if the vertical wires could be used to route within a column anywhere in the chip, not just at the borders, and in fact such an extension is under consideration. This is about as compact a Triptych layout as can be expected for a random logic function.

```
INORDER = s1 s2 d1 st SB0 SB1;
OUTORDER = NSB0 NSB1 r1 y1 g1 r2 y2 g2 sd;
NSB1 = !st * !g1 * !g2;
y1 = r2 * 51;
g1 = r2 * !51;
r2 = !st * SB0 * !9 + !st * !SB0 * 9;
y2 = 53 + 45;
g2 = !st * !r2 * !y2;
sd = 12 + 45;
51 = s2 * !SB1 + !SB0 * SB1;
9 = !SB1 + !d1;
45 = s1 * !SB1 * 46;
52 = !d1 * SB1
53 = 52 * 46;
12 = !SB1 * 18;
NSB0 = !st * !r2;
46 = !st * SB0;
18 = !SB0 * s2 * !st;
```

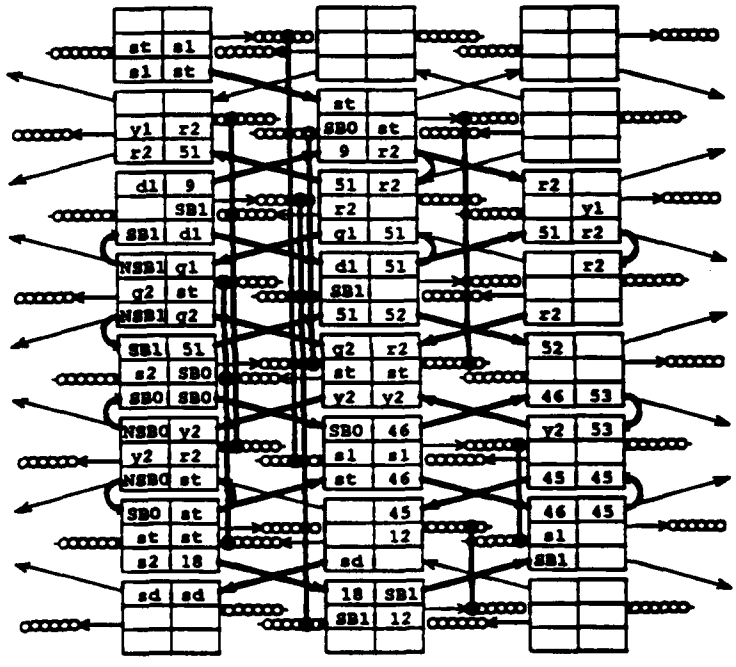


Figure 8 Factored equations and Triptych realization of the traffic light controller.

Lyon bit-serial multiplier

Although our experience shows that Triptych can be used to implement a wide range of circuits, its locally connected structure makes it especially good for repetitive arrays like bit-serial arithmetic circuits. The Triptych structure has some of the same features (e.g., nearest neighbor connections) as the Labyrinth FPGA which was targeted to bit-serial and pipelined/systolic circuits. We have chosen the Lyon bit-serial multiplier cell (Lyon 1981) shown in Figure 9 as a representative circuit from this class. A full n-bit multiplier comprises n copies of this cell, and signal processing circuits typically make use of several of these multipliers, containing many individual cells.

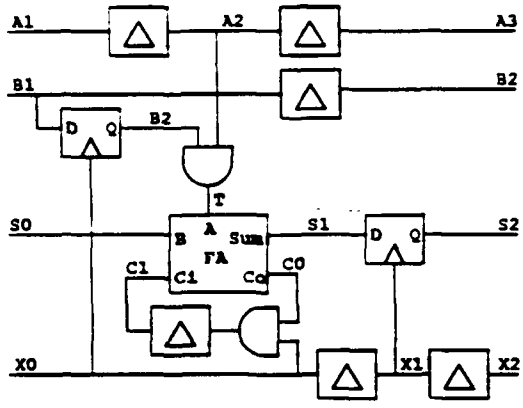


Figure 9 Design of a single Lyon bit-serial multiplier cell.

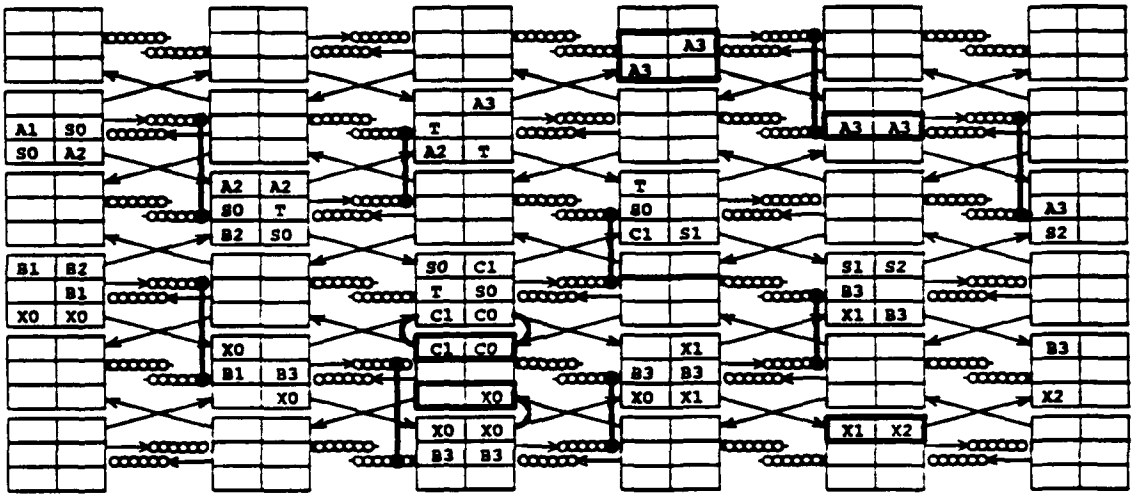


Figure 10 Layout for the Lyon bit-serial multiplier cell.

This multiplier cell presents the same classic layout problem as that faced by VLSI cell designers. The cells need to tile horizontally so that inputs match outputs and vertically so that little space is wasted between adjacent multiplier cells (see Figure 10). In this case, however, there is an extra dimension since a string of multiplier cells will wrap around the chip on the opposite direction of RLBs. Since there is one RLB that is used from the opposite direction (position E-2), the layout must provide a "hole" into which this RLB can fit (position B-4). Note that these two logical RLBs can share a single physical RLB since they use independent paths through the RLB. The cost of this multiplier cell design is 12.5 RLBs which is not much more than the smallest conceivable design, which costs 11 RLBs. The 0.5 RLB results from the sharing of one RLB (positions A-3 and F-4) between two vertically adjacent multiplier cells.

Measurement and comparison

Although our experience with mapping circuits to Triptych is thus far very limited since automated placement and routing are still being developed, we have some preliminary measurements of the cost of Triptych implementations relative to Labyrinth and Xilinx. Since

the area cost is measured for each FPGA type in terms of the number of logic blocks used for that technology, we must first normalize the cost of the different FPGA logic blocks to be able to compare the different FPGAs. Although such relative figures are difficult to come by, we have combined a relative size estimate based on die size and number of logic blocks, along with the relative number of program bits to arrive at the following relative cost figures. Using the cost of the Labyrinth logic block as the normalized unit cost, we estimate that the cost of a Triptych RLB is about 4-5 (4.5) units and that of a Xilinx CLB (configurable logic block) is about 20-25 (22) units. This places the Triptych logic cell squarely in the middle between the very cheap Labyrinth cell and the relatively expensive Xilinx cell.

Table 3 gives the approximate cost of implementing a number of circuits using all three FPGAs, both in terms of each technology's logic blocks and in normalized cost as defined above. We believe these figures indicate that Triptych is a promising architecture for a range of different circuits. These results are of course very preliminary and many more experiments must be done with other circuits and using automatic place and route tools.

Circuit	Labyrinth # blocks	normalized cost	Xilinx # blocks	normalized cost	Triptych # blocks	normalized cost
Multiplier	54	54	5	110	12.5	56
Traffic	280	280	6	132	24	108
s208	N/A	N/A	26	572	61	275

Table 3. Results of mapping three examples: the Lyon bit-serial multiplier; a traffic light controller; and ISCAS benchmark s208 the Labyrinth, Xilinx and Triptych.

Issues in mapping to Triptych

We have successfully mapped a number of regular structures and small control circuits to the Triptych architecture, and we are currently working on CAD tools that will automatically perform the mapping for arbitrary circuits. As with other FPGAs, the process of mapping a circuit onto Triptych can be considered to consist of three steps:

- covering: forming a circuit graph containing function nodes with at most three inputs,
- placement: assigning these function nodes to cell locations on Triptych, and
- routing: making the connections in the graph through the available routing on Triptych.

If the circuit to be mapped has a regular structure, as is encountered in domain-specific applications such as digital signal processing, an initial pattern for the repeating portion may be derived by hand. Circuits without regular structure, or "random logic", must rely on heuristic-based automatic placement and routing methods similar to those used by other FPGAs. However, because Triptych's routing resources are highly constrained, placement and routing must be more closely integrated than they are in other FPGAs.

For the covering portion of mapping to Triptych, we assume that a tool such as *chortle* or *mis-pga* is available to express the original circuit as a graph of elementary gates and then cover the graph's fanout-free trees with collections of three-input RLBs (Francis 1991, Murgai 1990). It should be noted, however, that a covering which minimizes the total number of RLBs may not be optimal when placement and routing are taken into consideration. For example, if after placement two of the inputs to a three-input RLB naturally both occur at a

single location distant from that RLB, it is usually advantageous to split the RLB into two two-input functions. If this is possible, we can route one less signal across the large distance. Clearly, such situations are not unique to Triptych. However, we particularly wish to avoid routing extra signals horizontally whenever it can be avoided. Otherwise, RLBs become congested with signals they do not use. Such optimizations are difficult to predict at cover time and thus need to be attempted during routing.

Because Triptych's routing resources are limited and fairly tightly constrained, we believe it is necessary to keep placement and routing well integrated. Evaluating possible placements with simple measures of routing length can lead to placements whose congestion make routing nearly impossible. Currently, we are exploring iterative improvement methods for placement which will assign an RLB only into locations which are adjacent to enough free tracks to route the RLB's inputs and outputs. Thus, we avoid congestion at a local level whenever we place an RLB.

A complicating factor is that Triptych's distance metric is non-symmetric. All pairs of RLBs that face in the same direction, except those in the same column, have a distance from the first's output to the second's input different than that of the second's output to the first's input. Also, vertically adjacent blocks have the same routing distance as diagonally adjacent blocks. For these reasons, routing distance is not well represented by the x-y coordinates given to the RLBs. Work is ongoing to develop an integrated force-directed placement procedure, a Triptych-specific distance measure, and the congestion avoiding method mentioned above.

CONCLUSIONS

The new FPGA architecture presented in this paper was motivated by three needs: permitting the realization of delay-critical circuits; including data-path and control elements in the same array; and minimizing the space devoted to routing resources. We believe that Triptych achieves these goals given the experience gained so far with many example circuits; a few of which have been presented above. The examples have proven to be more densely packed and to have delay characteristics comparable to the other FPGAs.

The most interesting and challenging future direction for research is automatic mapping. Triptych requires that the functional and interconnect elements not be treated separately. Combining the considerations for covering, placement, and routing should allow us to develop mapping tools that more precisely predict the performance of circuits and more accurately trade off density for speed.

Pedagogically, the design of a field-programmable gate array made an excellent class project. It exposed our students to a large vertical slice of the design problem stretching from electrical details to technology mapping issues. They were able to experience many of the issues and tradeoffs that must be resolved in both integrated circuit and logic design. Furthermore, the design and layout were ideal for a class project because the work was easily partitioned and only a small number of cells needed to be considered. In this respect, the Triptych effort was a resounding success and has motivated several of the non-VLSI, non-CAD oriented students to continue to look into VLSI issues.

In summary, we learned much from the design experience and believe we have a viable new FPGA architecture for circuits where either minimization of delay is of critical importance and/or data-path elements must be included with control logic. There is much work to be done,

especially in the area of automatic mapping, and promising directions are just beginning to be pursued.

ACKNOWLEDGEMENTS

Thanks to David Hubbell, Daniel Kerns, Alexander Klaiber, Dylan McNamee, J. Scott Penberthy, Radhika Thekkath, and especially Christopher Hébert for participating in CSE568 and contributing to the early design phases of the Triptych layout and mapping techniques. Thanks also to Rick Hood for the Labyrinth measurements in Table 3 and help developing the relative cost measures. This research was funded in part by the Defense Advanced Research Projects Agency under Contract N00014-J-91-4041. Gaetano Borriello and Carl Ebeling were supported in part by NSF Presidential Young Investigator Awards with matching funds provided by IBM Corporation and Sun Microsystems. Elizabeth Walkup holds an NSF Graduate Fellowship.

REFERENCES

Actel Corporation, "ACT Family Field Programmable Gate Array Data Book", 1991.

W. Carter et al., "A User Programmable Reconfigurable Gate Array", Proceedings of the IEEE Custom Integrated Circuits Conference, May 1986.

K. A. El-Ayat, A. El-Gamal, R. Guo, J. Chang, R. K. H. Mak, F. Chiu, and E. Z. Hamdy, "A CMOS Electrically Configurable Gate Array", IEEE Journal of Solid-State Circuits, Vol. 24, No. 3, June 1989, pp. 752-761.

R. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs", Proceedings of the 28th Design Automation Conference, June 1991.

F. Furtek, G. Stone and I. W. Jones, "Labyrinth: A Homogeneous Computational Medium", Proceedings of the IEEE Custom Integrated Circuits Conference, May 1990.

R. F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing", Proceedings of VLSI'81, August 1981.

R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, A. Sanggiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays", Proccedings of the 27th Design Automation Conference, June 1990.

J. Rose and S. Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays", IEEE Journal of Solid-State Circuits, Vol. 26, No. 3, March 1991, pp. 277-282.

J. Rose, R. J. Francis, D. Lewis, and P. Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency", IEEE Journal of Solid-State Circuits, Vol. 25, No. 5, October 1990, pp. 1217-1225.

S. Singh, J. Rose, D. Lewis, K. Chung, and P. Chow, "Optimization of Field-Programmable Gate Array Logic Block Architecture for Speed", Proceedings of the IEEE Custom Integrated Circuits Conference, May 1990.

I. Sutherland and R. Sproull, "Logical Effort: Designing for Speed on the Back of an Envelope", keynote address at the University of California at Santa Cruz Advanced Research Conference in VLSI, March 1991.

Xilinx, Inc., "The Programmable Gate Array Data Book", 1991.

OEsim: A Simulator for Timing Behavior

Tod Amon and Gaetano Borriello

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract Digital circuit behavior consists of two components: functionality and timing. Most computer-aided design tools focus on functional aspects of behavior emphasizing data transformation and sequencing of operations. Timing aspects of behavior have received far less attention and have only recently come to the fore as concerns for system level simulation, system integration, and synthesis are becoming more acute. In this paper, we present a behavior simulator, called OEsim, that addresses these issues. The underlying model is based on OEgraphs and supports the specification of timing constraints at many levels of abstraction, from propagation delays to interface behavior. The simulator is an ideal design validation tool in that it supports incremental checking of timing constraints during simulation and minimizes the description of circuit details unnecessary to timing simulation. The key ideas are the use of causality for the specification of abstract constraints through the use of a restricted first-order predicate calculus that has a clear simulation semantics and an efficient realization in the simulator. OEsim has been implemented in C++ and constructs a compiled simulation from an OEgraph representation.

I. Introduction

Design representations capture many facets of digital circuit specifications. Circuit behavior is the high-level description of what a circuit does without overly specifying how that computation is performed. Circuit structure is the low-level description of how the computation is implemented, that is, the logic gates and flip-flops used. Functional aspects of both behavior and structure describe the data transformations and computations to be performed on the inputs in order to generate the outputs. In contrast, timing relationships for behavior and structure describe temporal properties such as minimum and maximum separation times for signal events. Figure 1 maps out the space of design representation schematically.

It is of critical importance that a design representation support user validation. Making sure that what was specified is what was desired is the first step in verifying a design and cannot be automated. A simulator provides the user with the capability to try out the circuit and make sure it behaves as expected (at least for a subset of all possible inputs). The ability to simulate the specification at any point in the design space is also crucial as designs may consist of both behavioral and structural components. Existing simulators focus primarily on functional or structural aspects and include little support for the simulation of timing behavior.

	Behavior	Structure
Functional	concurrent program abstract / high-level	register-transfer level descriptions
Timing	real-time constraints (context dependent)	propagation delays setup/hold times

Figure 1. The design representation space is divided betwee behavior and structure and orthogonally between functional an timing aspects. Examples in each of the four quadrant demonstrate the distinctions.

Detailed timing behavior is an important part of high level specification, simulation and synthesis. Whe synthesizing a circuit not all aspects of its behavior are unde the control of the designer. That is, the circuit must conform t the environment in which it will be placed. The environmer may demand that particular timing relationships be respecte These can be as simple as setup and hold times or as complex a the spacing between messages sent over a local area networ Within a circuit, we must be able to adequately describe th timing methodologies used to implement different parts of th circuit. These include precharging constraints, pipelin interlocks, and multiple-phase clocks. Also, we must be abl to provide an abstraction of a circuit based on its interfac behavior, a crucial capability for information hiding[1] an modularity.

We have developed a new representation which support the specification of timing behavior and was designed wit simulation and incremental synthesis in mind. A clea simulation semantics was a requirement for all the features c the model enabling, among other capabilities, incrementa timing constraint checking during simulation. We hav implemented a simulator, based on our representation, whic provides empirical verification of timing behavior.

This paper is divided into five major sections. Section I motivates our approach to timing specification and provide the details of our new model. Section III describes ou simulator and how simulation efficiency concerns affected th development of the representation. Section IV contains a larg example which demonstrates the usefulness of th representation and the simulator. Section V contains comparison of the simulator with existing high-leve simulators and concludes the paper by summarizing th contributions.

II. A New Model for Timing Specification

To describe timing relationships between circuit events w need to identify the circuit events being constrained an express properties which the events need to satisfy. Fo example, a setup constraint applies to two circuit events (e.g. an event on an input and the next rising clock edge), an requires that they be separated by a fixed amount of time (e.g.

[time of edge - time of input] > 5ns.). Circuit events are usually constrained by specifying separation times — both absolute ("5ns") and relative ("3 cycles") to characterize propagation delays, clock rates, etc. At higher behavioral levels, timing constraints are more abstract because they specify separation times between abstract behavioral events. Constraints may apply only in a specific behavioral context (e.g., during read and not write operations) or as a result of a specific causal path. Typically timing constraints are informally specified using timing diagrams and tables (see Figure 2).

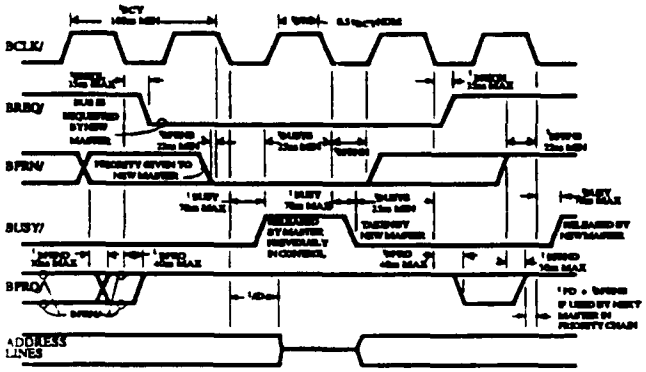


Figure 2. A timing diagram from the specification of the Intel Multibus [1]. Constraints specify separation times between events (i.e., changes in logic level on signal wires).

To identify circuit events both chronological and causal relationships are needed. Chronological relationships rely on time as a means of identifying the events being constrained (e.g., the *next* clock edge) and appear often in many representations (i.e., temporal logics [2,3], hardware description languages [4], waveform algebra [5], and many others). Causal relationships are equally important. For example, in Figure 3, requests to a non-FIFO queue need to be acknowledged in less than 100ms. However, in this case, chronology cannot be used to pair up and identify the request and acknowledge events as they may occur in a different order on the output than on the input (i.e., a later request may be acknowledged before an earlier one).

Figure 3. A non-FIFO queue that acknowledges requests in a different order than the requests were received.

II.1 The OEgraph Model

The representation we have proposed [6,7] is a straightforward graph model whose basis consists of two types of nodes connected by directed arcs to form a bipartite graph. A restricted predicate calculus is used to express timing constraints. The model contains a novel concept: event ancestry, to permit reasoning about causal as well as chronological relationships. We present a summary of the model omitting many of the features that are not directly related to the expressibility of timing behavior (the applicability of the model to the entire design space is discussed in detail in [7]).

Event nodes represent changes in logic level on circuit wires. If an event node is on a cycle in the graph then it may occur multiple times. Therefore, it is important to distinguish a *discrete* event from an event node. An event node represents an event that may occur; a discrete event is an actual occurrence of that event. *Operation* nodes or *boxes* are the second type of node in our graph representation and correspond to the functional aspects of behavior and structure. Boxes contain program code (e.g., C++ source code) that is evaluated whenever an input event occurs. The evaluation may conditionally generate output events which will occur at some future point in time and/or possibly change internal state. An example of an operation node is a logic gate that may generate an output event whenever an input event occurs. The delay in generating the output event corresponds to the propagation delay of the gate. The delay is specified by either a fixed value or a range of values and a distribution function to indicate uncertainty with respect to when the event will actually occur (e.g., the delay is dynamically computed each time a discrete event is generated). A more abstract example of a box is one that forks two independent processes: an input event arrives at an operation node that will then cause two parallel output events thereby permitting two parts of the specification to proceed in parallel. The events, in this case, do not correspond to logic transitions and instead represent abstract control flow.

Dependency arcs specify the flow of events in to and out of boxes. The graph is bipartite because dependency arcs specify flow of control by directing events into boxes and the output of boxes to events. Events have an in-degree of at most one dependency arc, thus each event is either an external event or is caused by the execution of a single operation node. Events may have arbitrary out-degree.

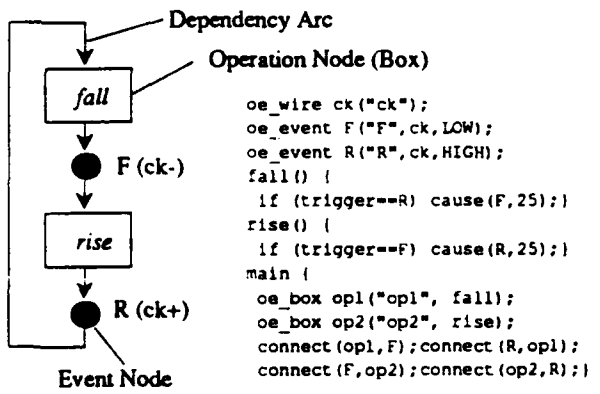
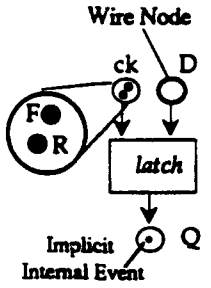


Figure 4. Graphical and textual versions of a simple single phase clock in the OEgraph representation (from a behavioral perspective). The clock has a cycle time of 50 time units with a 50% duty-cycle.

Event nodes that affect changes on the same wire can be grouped into an event node set. Since every possible change on the wire (every discrete event) is collected into the set we refer to such a set as a wire. In practice, wires can be considered a third type of node in our graph. Operations may have wires as inputs (any event that is a member of the wire is an input to the operation) and may also have wires as outputs (the output wire's value is changed via an implicit internal event). An operation can ask about the value of an input wire (i.e., the effect of the most recent discrete event on that wire).

The model contains additional elements for handling buses and other structural constructs.




```
oe_wire ck("ck");
oe_wire D("D");
oe_wire Q("Q");
latch() {
 if (trigger==ck && ck==HIGH)
  cause(Q, uniform_delay(5,10),
        valueOf(D));}
main {
 oe_box op1("op1", latch);
 connect(ck,op1);
 connect(D,op1);
 connect(op1,Q);}
```

Figure 5. A clocked edge-triggered D-latch in our representation. Note that the propagation delay is uniformly distributed between 5 and 10 time units.

The model uses an event based paradigm because timing constraints are relationships between circuit events and are more easily expressed in an event based model [8]. With respect to our representation, timing constraints are *relationships between discrete events* — not event nodes. For example, the setup time constraint on the D-input to the latch of Figure 5 applies to any event on D and the next chronological occurring rising edge event on the clock. Chronological relationships can often be used to identify the discrete events involved in a constraint. However, constraints may be relative to a particular execution path in a complex graph, and constraint specification must include a way of getting at *this* history. Causal relationships also need to be described and reasoned about. An example of a constraint requiring a causal relationship is a response time constraint on requests to the queue of Figure 3. Causality pairs up corresponding request and acknowledge events so that a maximum separation can be specified.

We have developed the concept of event ancestry to address the specification of causal relationships. An ancestor of a discrete event is any previously occurring discrete event that led to the generation of its descendant through its effect on a series of boxes. Thus, every discrete event has an ancestry tree, consisting of its immediate ancestors and their ancestors, transitively. Whenever an operation decides to generate an event, the new discrete event has as ancestors the most recent discrete occurrences of the input events named as ancestors (all inputs by default). When an input wire is named as an ancestor, the most recent discrete event on the wire (as seen by the operation) is the ancestor. Internal state is treated the same way as output events in that internal state variables have ancestors and can be named as ancestors of output events. When a state variable is named as an ancestor all of the ancestor events of the state variable are ancestors of the discrete event. This permits decomposition of operations.

The language used for the specification of timing constraints (which are assertions about the desired behavior of the specification) is based on a first-order predicate calculus that consists of:

- standard Boolean and integer functions and relations ($\vee$, $\wedge$, $\rightarrow$, $\sim$, +, −, *, /, <, >, =)
- quantification of discrete events and a test for equality (existential "$\exists$", universal "$\forall$", and equality "=")
- a relation "$\in$" to test if a discrete event is an occurrence of a named event or a member of an event set
- an integer function "τ" that returns the time at which a discrete event occurs
- a relation "anc" to test whether a discrete event is an ancestor of another
- a function "v" returning the wire value for a discrete event
- associated wire value relations

The time at which an event occurs is not an infinite precision real, but instead is an integer. Thus multiple events may appear to occur at the same time, due to a natural granularity problem that also exists in the physical world (i.e., at some level of detail it is not possible to determine which of two events actually occurred first). This affects the definitions of the primitive relations described later in this section.

The full first-order predicate calculus introduces problems (discussed in the next section) which we have addressed by restricting the representation of timing constraints to the following format: 1. universal quantification of the discrete events involved in the constraint, 2. specification of the context within which the constraint must hold and, 3. specification of a particular timing relationship that is required to be true when the context is true.

quantified discrete events in a context	$\Rightarrow$	timing requirement for the discrete events

In order to capture much of the expressibility of the full calculus, while restricting it to the format above, we added the following three relations (lower case variables represent quantified discrete events).

- **mra(x,S,y)** to test whether x's most recent S ancestor (of all the events in the set S) is/might be y.

$mra(x,S,y) \equiv \forall z\ (anc(x,y) \wedge (z \notin S \vee \sim ancestor(x,z) \vee \tau(z) \leq \tau(y)))$

- **pco(x,Y,y)** to test whether x's previous chronological occurrence of an event in the set Y is/might be y.

$pco(x,Y,y) \equiv \forall z\ (y \neq x \wedge \tau(y) \leq \tau(x) \wedge (z \notin Y \vee \tau(z) \geq \tau(x) \vee \tau(z) \leq \tau(y))$

- **nco(x,Y,y)** to test whether x's next chronological occurrence of an event in the set Y is/might be y.

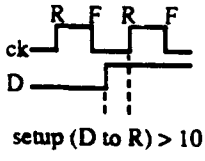
$nco(x,Y,y) \equiv \forall z\ (y \neq x \wedge \tau(y) \geq \tau(x) \wedge (z \notin Y \vee \tau(z) \leq \tau(x) \vee \tau(z) \geq \tau(y))$

These three relations are formally defined using the full calculus and encapsulate, in a more efficient and compact form, concepts which are essential for timing constraint expression. All constraints consist of combinations of these relations and the logic primitives.

II.2 Examples

Figure 6 contains the specification of the setup constraint for the D-input to the latch of Figure 5. The textual representation for the calculus is very straightforward. Constraints (and discrete events) are declared and specified using the three part format described above. Constraints can be specified individually as shown in Figure 6 or through the use of subroutines that can be parameterized (as is the case with common constraints such as setup and hold).

Most constraints imposed on a circuit have a simple semantics and are thus easily expressed. For example a constraint specifying that an input waveform has a minimum pulse width is easily expressed: quantify two edges of the waveform, and if they are chronologically related (e.g., pco(edge1,waveworm,edge2) then they must be separated by the minimum pulse width (e.g. τ(edge1)-τ(edge2)$\geq$ minpulse).



... inserted into main( ) of figure 5

```
oe_constraint setup("setup");
discrete_name D0("D0",D);
discrete_name R0("R0",R);
setup.quantify(D0,R0);
setup.context(nco(D0,R,R0));
setup.require(timeOf(R0)-timeOf(D0) >10);
```

restricted calculus:

$\forall r\ \forall d\ (\ (\ r \in R \wedge d \in D \wedge nco(d,R,r)\ ) \rightarrow (\ \tau(r) - \tau(d) > 10\ )\ ) \equiv$

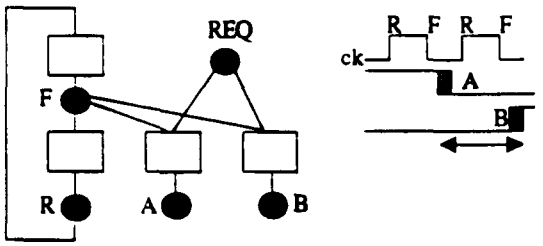
full calculus:

$\forall r\ \forall d\ \exists z\ (\tau(r)-\tau(d)>10 \vee r \notin R \vee d \notin D \vee r=d \vee \tau(r)<\tau(d) \vee (z \in R \wedge \tau(d)<\tau(z)<\tau(r)))$

Figure 6. Our representation of the setup constraint for the D input to the latch in Figure 5.

Many constraints which appear to be simple in nature actually have a complicated semantic meaning. For example a constraint stating that "two events occur one cycle apart" is subject to many interpretations. Timing diagrams and tables attempt to convey the semantics of such constraints but are informal specifications and are often insufficient. Using OEgraphs and the constraint specification language described above we can formally specify such a constraint.

Figure 7 contains a constraint that states that events A and B are required to occur exactly one cycle apart whenever A and B occur from the same request event (REQ). The constraint is complicated by the fact that A and B occur synchronously to the falling edge of the clock (F) and may occur anywhere within the shaded region shown in Figure 7 (requiring an exact separation time between A and B would thus be incorrect). This is another example of a constraint that cannot be expressed without the use of event ancestry. For example, waveform algebra [5] cannot be used to describe this constraint because it cannot pair up the A and B events to be constrained.



quantify: A0, B0, F0, F1, REQ1

context: pco(A0,F,F0) ∧ pco(B0,F,F1) ∧
mra(A0,REQ, REQ1) ∧ mra(B0,REQ,REQ1)

requirement: pco(F0,F,F1) ∨ pco(F1,F,F0)

Figure 7. A sequential logic constraint requiring two events to be one cycle apart.

The context for the constraint establishes the identities of the quantified events. F0 is the clock edge prior to A0; F1 is the clock edge prior to B0; A0 and B0 share the same REQ ancestor. The requirement is simply that F0 be the clock edge prior to F1 or that F1 be the clock edge prior to F0.

III. Simulation

As mentioned in section II the representation for timing constraints is a restricted version of the full first-order predicate calculus. In particular, the calculus only supports universal quantification where all quantifiers precede all clauses. Existential quantification and the ability to negate quantifiers was completely removed.

These simplifications were made for a number of reasons. In simulation, the universe of discrete events changes as new events occur and are added to the universe. Constraints are statements about the infinite universe of events. If the universe is constantly changing, when can constraints be checked and violations reported? Violations can be detected and reported for constraints with universal quantification because a particular instantiation of discrete events causes constraint violation. With existential quantification violations often can never be reported because the events that satisfy the constraint may not yet have occurred. Of course, we can report satisfaction for existentially quantified constraints and not for universal ones, but in simulation (i.e., for validation purposes) constraint violations are of primary interest. Often we would not be able to conclude anything about constraints that contain both quantifications and the simulator would be extremely inefficient because constraints would be repeatedly checked.

In addition, event generation (existence) is represented in the functional components of the graph. If an event is required to exist given a particular set of circumstances, an operation can be defined which generates the event given the circumstances. Therefore, existential quantification — which is used primarily to describe functionality, is already encapsulated within our representation.

Of course, there are some relationships that can be easily expressed in the full calculus which do not suffer from the problems described above. We added pco, nco, and mra to our restricted calculus because they encapsulate relations which are essential for timing constraint expression. They have semantics that are easily implemented by the simulator and the three relations do not introduce complications which would prevent efficient incremental constraint checking — the important motivation for our restrictions. Of course restricting the calculus does weaken its expressibility. However, based on experience with many example specifications we believe that there is no effect with respect to the specification of timing requirements encountered in digital systems.

III.2 Simulation Efficiency

The main efficiency concern in the simulator is the incremental checking of timing constraints. Whenever a new discrete event occurs, all constraints that quantify the event must be checked. The new event is quantified in the constraint and all possible combinations of discrete events (that previously occurred) are tried for the other quantified events. This mechanism ensures that each constraint will only be checked once for each possible assignment of unique discrete events. For example, consider the following constraint: "quantify: X_0, Y_1 context: $\tau(X_0)>\tau(Y_1)$ and some requirement". If X events had previously occurred at time 5 and 12 and a new Y event occurred at time 17, the constraint would be checked at time 17 with Y@17 instantiated for Y_1 and with X@5, and then X@12 instantiated for X_0. If a new X event occurred at time 20 the constraint would be checked with X@20 instantiated for X_0 and with Y@17 instantiated for Y_1. Constraint checking requires an exponential amount of time with respect to the

number of events quantified by a constraint. Fortunately, most timing constraints contain only two quantifications. This is because most constraints require a separation time between two events, and a simple context (i.e. a simple chronological or causal relationship) is sufficient to identify the discrete events being constrained.

Many other optimizations are performed to make the simulator more efficient. The occurrence of an event need not always trigger the evaluation of a constraint quantifying that event as described above. If a constraint's context strictly orders the time of occurrence of two events then a new event need not be instantiated into the earlier event because it is known that the context will be false (the later event has not yet occurred). For example, since $\tau(X_0)>\tau(Y_1)$ new occurrences of Y do not require checking of the constraint. This static optimization is also applied to constraints which refer to ancestry because an ancestor must always occur before its descendant. Similar optimizations are made for constraints involving the chronological relations (pco and nco) and for events which are known to occur at the same time. In this case, only one of the events needs to trigger constraint evaluation.

The optimization described above helps reduce the amount of time required by constraint checking. However, as the simulation progresses and new events occur, constraint checking becomes more time consuming because more events are instantiated each time a constraint's context is evaluated. This problem is related to another efficiency concern: the amount of space required to store past events and maintain the directed acyclic ancestry graph. This is particularly troublesome in that larger simulations usually generate lots of events, and some events (e.g. clock edges) occur very frequently. However, it should be pointed out that larger simulations (i.e. large OEgraphs) are not inherently less efficient to simulate. Timing constraints apply to small numbers of events, irrespective of graph size, and the efficiency of the simulation engine is related to the amount of parallelism inherent in the graph, not the graph size.

One approach to this problem is discrete event removal. Discrete events can be removed from storage if it can be shown that they are no longer needed for constraint checking. In the simple case, if an event is not a part of any quantified set, it need not be stored. Before removing the discrete event, the ancestry information which is contained in the discrete event node needs to be pushed outward. This is accomplished by connecting the children of the node to the parents of the node before deleting the node. We intend to extend this simple optimization (which has been implemented) to support the removal of discrete events even when they are quantified and appear in timing constraints. Many constraints involve simple chronological relationships that do not require storing complete histories (e.g. instead of storing every clock edge, only the two most recent edges are stored since they are the only ones involved in constraints) and it should be possible to determine a priori how many events need to be stored. With respect to causality, often only the more recent causal chains are important, and in this case an event removal technique akin to garbage collecting could operate periodically and effectively prune the ancestry DAG.

For a given constraint, if analysis can prove that under any possible execution of the specification the context is always false, the constraint would not need to be checked. Likewise, the constraint would not need to be checked if any possible execution would result in the requirement always being true. Such a simulation optimization tool (capable of detecting these two conditions) would constitute a very powerful verification tool.

III.2 Implementation and Interface

OEsim is a compiled simulator that takes as input a C++ description of an OEgraph and its associated constraints and generates a compiled and linked form that includes the simulator front-end. This produces a single executable simulation program. By virtue of being a compiled C++ program, an operation node's program can include arbitrary C++ code that can be used to provide special interactions with the user (e.g., read and write data files). The simulation steps through the execution of the OEgraph and contains commands which support user control (e.g., single stepping, setting breakpoints, scheduling events, etc.). Figure 8 contains an example simulation showing a violation of the setup constraint described in Figure 6.

```
Welcome To Simulation v1.3, Mon Nov  5 15:13:43 1990
... no stimulus file (fig6.itf) found
oesim-0> schedule-event F 0
oesim-0> schedule-wire D HIGH 60
oesim-0> schedule-wire D LOW 120
oesim-0> run-to 150
event_occurs at time:    0 event F
event_occurs at time:    25 event R
event_occurs at time:    50 event F
event_occurs at time:    60 event D$<external>(D*HIGH)
event_occurs at time:    75 event R
event_occurs at time:    100 event F
event_occurs at time:    120 event D$<external>(D*LOW)
event_occurs at time:    125 event R

 ** Constraint Violation: setup:
R0 D0 : nco(D0, R, R0) ==> ((t(R0) - t(D0)) >10)
 R0 =  unique event: R occurrence: 3 at time: 125
 D0 =  unique event: D$<external> occurrence: 2 at
time: 120

 stopped at time: 150
```

Figure 8. An example simulation showing a violation of the setup constraint described in Figure 6.

The simulator was implemented using C++ in a UNIX environment and has already been instrumental in debugging specifications. We have used our simulator to describe a wide range of examples derived from real circuits or extracted from the specification and synthesis literature—the largest being a partial specification and simulation of the Intel Multibus (see [7]). We have yet to analyze the performance of OEsim in detail, but have found it to be efficient and capable. Compile time (a few minutes) has always exceeded simulation time except for un-optimized simulations containing constraints that quantify many events. Space efficiency has not been a problem (i.e. storing over a million events) but further work is needed to support larger and more lengthy simulations.

At the University of California at Berkeley, OEsim is being used to represent the abstract interfaces of complex components that must be interconnected on a printed-circuit board or multi-chip module. The result is a simulation module for the glue logic that must be designed and an understanding of the many timing relationships that must be maintained when the components are interconnected.

V. Conclusion

OEgraphs were designed with simulation in mind. A clear simulation semantics was a requirement for all the features of the model. An important goal consisted of modeling general

timing constraints expressed using both chronological and causal relationships. The difficulty with timing constraints was ensuring that the calculus used for their specification had a clear simulation semantics and would support incremental constraint checking. This was accomplished with the three new primitives and format restrictions outlined in section II.

Existing simulators (i.e., THOR [9], COSMOS [10], VHDL [11]) focus primarily on functional aspects of both behavior and structure. To consider timing constraints, users have to develop functional modules to check timing properties and report their satisfaction/violation. This is precisely the approach taken in VAL which uses a VHDL simulation engine and augments the VHDL with assertions in waveform algebra [12]. It is also important that these types of structured approaches are used rather than the more common ad hoc methods. Specification of timing constraints is very error prone and must be done consistently to be useful.

There are many reasons for developing a new representation and simulator instead of attaching our constraint language to a VHDL simulator. First, complete VHDL is very difficult to synthesize and we would have had to use a subset of the language. Many constructs in the language allow the manipulation of the simulation model and its event queues. It is not at all clear how to synthesize descriptions using these constructs. Second, to be able to refer to causal relationships among VHDL events would have meant modifying fundamental data structures to include the ancestry trees. Third, many aspects of behavior are best represented using an event model that is not completely supported in VHDL which is based primarily on a signal wire model. Lastly, we wanted the capability of including arbitrary C++ code in our simulations so that user interfaces and other interesting I/O could be directly incorporated in the executable for the simulation.

Implementation of OEsim is complete, although additional work to improve the user interface and to extend existing efficiency optimizations needs to be done. Our research emphasis has moved from OEsim to other tools which operate within the same framework. OEgraphs give time and timing constraints their long deserved equivalent status with structural and functional specifications. We believe the representation provides a target for the development of domain-specific description languages and a basis for incremental synthesis algorithms. The focus of our synthesis work is on control logic synthesis and scheduling algorithms. OEsim will be critical in verifying our efforts.

Acknowledgements

The authors gratefully acknowledge numerous discussions with Carlo Sequin and David Dill. Wayne Winder and Karen Bartlett were instrumental in the implementation of the simulator. Jane Sun at UC Berkeley has been daring enough to put our simulator to the test.

This work was supported by the National Science Foundation under a Presidential Young Investigator Award (MIP-8858782) and by the Defense Advance Research Projects Agency under contracts #N00014-88-K-0453 and #N00039-C-0182.

References

[1] Intel Corporation, *Intel Multibus Specification*, 1982.

[2] E. Clarke, E. Emerson, A. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, April 1986.

[3] B. Moszkowski, "A Temporal Logic for Multilevel Reasoning about Hardware", *IEEE Computer*, Vol. 18, No. 2, February 1985.

[4] J. Nestor, "Specification and Synthesis of Digital Systems with Interfaces", Technical Report CMUCAD-87-10, Dep't of Electrical and Computer Engineering, Carnegie-Mellon University, April 1987.

[5] L. Augustin, "Techniques for High-Level Synthesis and Specification: Waveform Algebra", *Fourth International Workshop on High Level Synthesis*, October 1989.

[6] T. Amon, G. Borriello, "On the Specification of Timing Behavior", *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau)*, August 1990

[7] T. Amon, G. Borriello, C. Séquin, "Operation/Event Graphs: A Design Representation for Timing Behavior", *Tenth International Symposium on Computer Hardware Description Languages and Their Applications (CHDL)*, April 1991.

[8] G. Borriello, "A New Interface Specification Methodology and its Application to Transducer Synthesis", Ph.D. Thesis, Computer Science Division, University of California at Berkeley, 1988.

[9] Thor Tutorial, VLSI/CAD Group, Stanford University.

[10] R. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits", *Proceedings of the 24th Design Automation Conference*, June 1987.

[11] M. Shahdad, et. al., "VHSIC Hardware Description Language", *IEEE Computer*, Vol. 18, No. 2, February 1985.

[12] L. Augustin, B. Gennart, Y. Huh, D. Luckham, A. Stanculescu, "An Overview of VAL", Technical Report CSL-TR-88-367, Stanford University, 1988.

SIZING SYNCHRONIZATION QUEUES: A Case Study in Higher Level Synthesis

Tod Amon and Gaetano Borriello

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract. In synthesizing a circuit from its description in a concurrent programming language, it is necessary to make decisions about how to implement synchronization constructs such as send and receive statements. The semantic model of these constructs is an infinite length FIFO queue that can handle all send events until they are paired up with corresponding receive events. In this paper, we describe an algorithm to size these synchronization queues while permitting the maximum parallelism between the communicating processes (circuits). It is an example of *higher* level synthesis in that the user does not include an explicit description of the queue in the specification as is necessary in current high level synthesis systems.

I. Introduction

High level synthesis is the process of deriving hardware implementations for circuits from concurrent programming languages or other high level specifications [1]. These specifications may include statements that are not inherently implementable. An example of this is the *send* and *receive* statements used for inter-process communication. These are especially important in the specification of digital signal processing, communications, and protocol circuits. The semantics of send and receive is that two processes are connected by an infinite length FIFO queue that can handle all send events so that they can be paired up with the corresponding receive event. In current high level synthesis systems, the user must explicitly bound the size of the queue and ensure that the two parallel processes (circuits) will never need to exceed that bound. This is normally achieved by the user placing additional control statements in the specification (e.g., in the form of handshaking signals) [2].

This leads to design style that is not modular and may also unduly limit the parallelism inherent in the specification. A *higher* level synthesis system should be able to use timing constraints on the rate of send and receive events to compute a bound for the queue size. As different interconnections are made between components it should not be necessary to change their internal specification (as is currently the case). If no bound is computable then the system could automatically modify the specification to include control logic to ensure that a certain size is not exceeded. The tradeoff between the size of the queue, the complexity of the control logic, and the parallelism in the circuit could be under the direction of the synthesis system and not the user. By sizing queues automatically, different tradeoffs can be made for different circuit configurations without changing the original specifications of the circuit components (processes).

Our model of a send/receive queue is shown in Figure 1. From a user's perspective, the queue is of infinite length — the user does not specify a size for the queue, it is assumed that the queue will be as large as needed to process every send event and store its data until a receive event indicates that the data should be output. From a synthesis perspective, there are two possibilities depending on whether the queue size can be bounded or not. An unboundable queue will necessitate explicit handshaking signals that can be used to block the producer and/or consumer processes (thereby restricting some possible parallelism). A bouded queue my be implement without explicit handshaking and if it is of a small enough size (i.e., one or two entries) may be removed altogether through a merging of the two processes. In this paper, we explore the problem of determining the size of a synchronization queue given constraints on the relative frequency of its send and receive events.

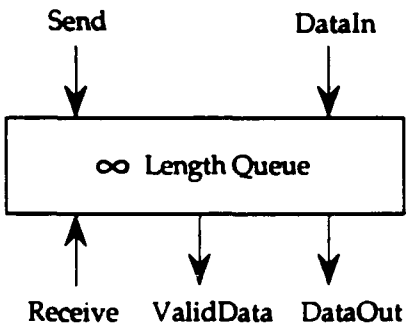


Figure 1. Our model of a generic FIFO queue. Send events indicate that data should be captured by the queue. Receive events indicate that the environment is ready to receive data. The queue issues ValidData events to indicate that it has taken data out of its FIFO and placed it on the output.

This paper is divided into four major sections. Section II describes the semantics of two particular queue models and their timing constraints. Section III describes a general technique for sizing synchronization queues. Section IV provides an example and Section V concludes the paper summarizing the contributions.

II. Queue Models and Timing Constraints

Many types of queues can be derived from the generic queue model presented in Figure 1. A *non-blocking* queue interprets receive events as requests for output which can be ignored if the queue is empty. The receive events do not block and arrive periodically to poll the queue for data. A *blocking* queue

interprets receive events as requests for output which must be satisfied — when the queue is empty a receive event blocks (no additional receive events will occur) until a send event arrives and the receive is acknowledged by ValidData. These two queues appear often in the context of digital signal processing, communications, and memory and system bus interfaces.

Asynchronous input data can be synchronized by a non-blocking queue (e.g., every falling clock edge corresponds to a receive event which indicates that data can be output and thus synchronized to the clock). In a blocking queue send and receive events are paired up; this corresponds to the send/receive constructs found in many concurrent programming languages.

The two queues and their timing constraints are formally defined using OEgraphs which provide a formal semantics and a framework for our analysis [3,4]. In this paper, we present a simplified syntax for timing constraints and rely on the informal descriptions of the queues presented above.

Timing constraints express information about the relative frequency of send and receive events. How quickly events can occur is specified using the constraint "$E_n \geq f$" which informally states that the n^{th} next occurrence of E (E_n) relative to any E event (E_0) takes place at least f time units after E_0. Thus, "$S_1 \geq 5$" states that send events (S = send, R=receive) are separated by at least 5. The constraint "$E_n \leq f$" specifies the slowest rate that events can occur. Thus, "$R_3 \leq 10$" states that the third receive event after any receive must take place within 10 time units.

The constraint definitions given above rely on being able to fully order the occurrence of events (e.g., to be able to talk about the 3rd receive event after another receive event). In OEgraphs the time at which an event occurs is not an infinite precision real, but instead is an integer [1]. Thus two events may appear to occur at the same time due to a natural granularity problem that also exists in the physical world (i.e., at some level of detail it is not possible to determine which event actually occurred first). Timing constraints express properties that are true given any interpretation for the order of events that occur at the same time (i.e., within the same time grain). The two constraints are more formally defined as follows:

> $E_n \geq f \equiv$ given n and f positive integers and two points in time t_1 and t_2, if there are n+1 or more Es in the closed interval $[t_1,t_2]$ then $t_2-t_1 \geq f$.
>
> $E_m \leq g \equiv$ given m and g positive integers and two point in time t_1 and t_2, if there are m+1 or more Es in the closed interval $[t_1,t_2]$ and less than m Es in the open interval (t_1,t_2) then $t_2-t_1 \leq g$.

Our analysis would be conceptually easier if we assumed that every receive or send event was separated by at least 1 time unit. However, we consider the more general case in which two events can appear to take place at the same time. This is important because our underlying semantics is based upon OEgraphs. Also, if more complicated queues (e.g., multiple input ports) are modeled as simple queues then the ability to handle inputs which appear to arrive at the same time is crucial.

More than one constraint may be available for a given event (e.g., $S_1 \geq 2$ and $S_5 \geq 30$). In some cases, one of the constraints may be redundant (e.g., if $S_1 \geq 2$ then $S_3 \geq 4$ is redundant because $S_3 \geq 6$ can be inferred from $S_1 \geq 2$) because one or more other constraints can be used to derive a more restrictive constraint. Redundant constraints can be removed but our results are not altered in the presence of redundant constraints and it is more efficient to process them than it is to detect and remove them. In some cases, the constraints will be ill-formed and the specification may be considered nonsensical (e.g., $S_1 \geq 5$ and $S_1 \leq 4$).

Our analysis is also based on an important assumption. "$E_m \leq g$" is formally defined as a timing constraint that is satisfied when E events occur; it does not state that the events must occur. Since we are interested in operating circuits, we assume that the events will necessarily occur, and are interested in the steady state behavior (i.e., we assume that receive events will not start occurring arbitrarily later than the send events).

It is important to note here that our usage of queues is quite different than in queueing theory. The constraints on events are not statistical averages but are abstractions of the propagation delays inherent in the hardware used to construct these circuits. As such, they are deterministic and enable us to set firm bounds on the size of queues and even consider their elimination by transforming the queue into other memory elements such as registers.

III. Queue Sizing

Queue sizing is accomplished by comparing the fastest rate at which send events can arrive with the slowest rate at which receive events must arrive. Before we present the algorithms for determining the queue size some preliminaries are in order including ascertaining that the constraints supplied are well-formed.

Intervals are specified using two integer time values and may be open (t_1,t_2) or closed $[t_1,t_2]$. Open intervals may be converted to closed intervals since times are integer values (e.g., $(t_1,t_2) = [t_1+1,t_2-1]$). We denote the maximum number of E events that can occur in the closed interval $[t_1,t_2]$ as $mostEin[t_1,t_2]$ and similarly denote the least number as $leastEin[t_1,t_2]$. Note that mostEin and leastEin are based on the size of the interval (e.g., t_2-t_1) not the actual values of t_1 and t_2. Regardless of any constraints on E events if $t_1>t_2$ then $mostEin[t_1,t_2] = 0$ and $leastEin[t_1,t_2] = 0$. In the absence of any constraints on E events and if $t_1 \leq t_2$ then $mostEin[t_1,t_2] = \infty$ and $leastEin[t_1,t_2] = 0$. The constraints on an event are ill-formed if there exists an interval in which the minimum number of events constrained to exist by a constraint is greater than the maximum number of events allowed by another constraint.

Theorem 1: Given any number of constraints, if $g/m < f/n$ (for some $E_n \geq f$ and $E_m \leq g$ constraint) then the constraints are ill-formed.

Proof: Consider an interval of size $fg - 1$, it must have at least ng Es (from $E_{ng} \geq fg$ which is derived by multiplying the constraint $E_n \geq f$ by g) and can have no more than mf Es (from $E_{mf} \leq gf$ which is derived by multiplying the constraint $E_m \leq g$ by f). But $ng < mf$, a contradiction which indicates that the constraints are ill-formed.

Given a set of well-formed constraints for both the send and receive events of a queue it is then possible to determine if the queue size is bounded or not.

Theorem 2: Given any number of constraints on S and R that are well-formed:

if $Max_{\{all\ R_m \leq g\}}(m/g) \geq Min_{\{all\ S_n \geq f\}}(n/f)$
then the queue is boundable,

if $Max_{\{all\ R_m \leq g\}}(m/g) < Min_{\{all\ S_n \geq f\}}(n/f)$
then the queue is not boundable.

Proof: The justification for this theorem is rather intuitive. "$Max_{\{all\ R_m \leq g\}}(m/g)$" represents the slowest rate at which

receive events must arrive and "$\text{Min}_{\{\text{all } S_n \geq f\}}(n/f)$" represents the fastest rate at which send events could arrive. A necessary (but not sufficient) condition for bounding the queue is that on average the number of R events that occur is greater than or equal to the number of S events.

If the queue is bounded then for a given time interval, we contrast the maximum number of send events that may have occurred with the minimum number of receive events that must have occurred during the interval. The size of the queue is the maximum difference over all possible interval sizes. The algorithm for determining maximum queue sizes is given below:

```
I = maxQ = 0;
repeat
   Imax = mostSin[0,I]-leastRin(0,I)
   maxQ = Max(maxQ, Imax)
   I = I + 1
until (Imax ≤ 0)
```

The algorithm terminates when a negative or zero sized queue is achieved for a given interval. We don't need to consider larger intervals because somewhere in the larger interval the queue will be empty and the larger interval consists of smaller sub-intervals which have already been analyzed. The only problem with the algorithm as presented is that there is no guarantee that it terminates (in fact, it would not terminate if it were used on unboundable queues). When the rates of S and R events are equal, the algorithm may not terminate because the queue may never empty but still contain a boundable number of entries. For this special case, the loop termination condition changes to:

```
...
until (I ≥ least common multiple of the
   g in Max{all Rm≤g}(m/g) and the f in Min{all Sn≥f}(n/f))
```

The algorithm terminates after running for a fixed number of intervals. Larger intervals are not considered because the behavior of the queue follows a worst-case pattern which is cyclical.

The algorithms above rely on two functions mostIn and leastIn that return the number of events (maximum and minumum number, respectively) that can occur in an interval. Note that the minimum number in the open interval is subtracted from the maximum number in the closed interval. This is necessary to properly handle events that may occur in the same time grain. A numerical algorithm is required because the maximum and minimum over all intervals cannot be determined analytically due to the discontinuous nature of the two functions.

The values returned by mostIn and leastIn are dependent on the constraints on the events and the size of the interval. We now consider the definition of these two functions and begin with two simple theorems that apply when only one constraint on an event has been specified.

Theorem 3: Given $E_n \geq f$ then $\text{mostEin}[t,t+I] = n(\lfloor I/f \rfloor + 1)$.
Proof: If $I < f$ there can be at most n Es in $[t,t+I]$ because if there were n+1 or more Es, this would directly contradict the definition of $E_n \geq f$. If $I \geq f$ then $[t,t+I]$ consists of $\lfloor I/f \rfloor + 1$ subintervals of size $\leq f$ (i.e., $[t,t+f-1]$, $[t+f,t+2f-1]$, ... $[t+f\lfloor I/f \rfloor,t+I]$) each of which can have at most n Es.

Theorem 4: Given $E_m \leq g$ then $\text{leastEin}[t,t+I] = m\lfloor (I+1)/g \rfloor$.
Proof: Every interval of size g−1 (i.e., $[t,t+g-1]$) must have at least m Es, otherwise the first E (or one of the first if tied) that occurred before the interval has as its m^{th} next E an E after the interval, this would directly contradict the definition of $E_m \leq g$. The closed interval $[t,t+I]$ consists of $\lfloor I/g \rfloor$ subintervals of size $g - 1$ (i.e., $[t,t+g-1]$, $[t+g,t+2g-1]$, ..., $[t+g\lfloor I/g \rfloor,t+I]$) each of which has at least m Es, and one interval of size $I - g\lfloor I/g \rfloor$ which has at least m Es if $I - g\lfloor I/g \rfloor = g - 1$. The summation of $m\lfloor I/g \rfloor$ plus m if $I - g\lfloor I/g \rfloor = g - 1$ easily simplifies to $m\lfloor (I+1)/g \rfloor$.

We next consider cases in which there are multiple constraints specified for an event (including both minimums and maximums). The additional constraints provide further information which may tighten the bounds on the minimum and maximum number of events that can occur in a given interval. For example, a constraint stating a minimum separation can actually increase the number of events that must occur in a given interval (e.g., given just $E_3 \leq 10$ an interval of size 5 is not required to have any Es, but if we add $E_1 \geq 2$ then the Es satisfying $E_3 \leq 10$ cannot occur together, and an interval of size 5 is now required to have at least one E).

Theorem 5: Given any number of $E_n \geq f$ constraints and any number of $E_m \leq g$ constraints then:

$\text{mostEin}[t, t+I] = \text{Min}_{\{\text{all } E_n \geq f\}}$ of:

$$n\lfloor I/f \rfloor + \text{Min}\left(\begin{pmatrix} n - \text{leastEin}[(I \bmod f)+1, f-1], \\ n, \qquad\qquad \text{if } I < f, \end{pmatrix}, \begin{pmatrix} \text{mostEin}[0, I \bmod f], & \text{otherwise} \end{pmatrix}\right).$$

$\text{leastEin}[t, t+I] = \text{Max}_{\{\text{all } E_m \leq g\}}$ of:

$$m\lfloor I/g \rfloor + \text{Max}\left(\begin{pmatrix} m - \text{mostEin}[(I \bmod g)+1, g-1], \\ 0, \qquad\qquad \text{if } I < g, \end{pmatrix}, \begin{pmatrix} \text{leastEin}[0, I \bmod g], & \text{otherwise} \end{pmatrix}\right).$$

Proof: (Omitted due to space constraints.) Intuitively, the two expressions are cross-coupled recurrence relations because a maximum (minimum) constraint can effect the spacing of events in an interval constrained by a minimum (maximum) constraint. We take the minimum (maximum) over all constraints so as to obtain the most constrained result. The basic expression sums the events in all sub-intervals of size f (g) and then takes the minimum (maximum) of n (0) and the number of events constrained to occur in the interval left over of size less than f (g).

We now present an algorithm (see below) to solve the cross-coupled recurrence relations and yield the maximum and minimum number of events in an interval of a given size, given any number of well-formed constraints on the events. It is based on two arrays (most and least) of size equal to the intervals in question. The arrays are initialized up to the size of largest of all the f's and g's (T). Then each constraint is applied to the arrays up to the size of the interval.

The initialization dominates the complexity of the algorithm which otherwise consists of a single application of

the constraints to the interval in question. Constraints are repeatedly applied until mostEin has been minimized and leastEin has been maximized. The complexity of the initialization is easily (albeit pessimistically) analyzed: After one iteration, most has an upper bound of $\text{Max}_{\{\text{all } E_n \geq f\}}(n\lfloor T/f \rfloor + n)$ and the least has a lower bound of 0 (from the initializations to 0 and infinity and one application of the constraints). Regardless of the number of iterations, most has a lower bound of 0 and least has an upper bound of $\text{Max}_{\{\text{all } E_m \leq g\}}(m\lfloor T/g \rfloor + m)$. Every iteration must either reduce most or increase least for some interval $\leq T$. This gives a very pessimistic upper bound on the complexity of the algorithm. The algorithm is linear with respect to the number of constraints and has constants that are polynomial with respect to the actual numbers specified in the constraints (i.e., cubic with respect to T). In practice, the algorithm has been observed to be nearly linear (with respect to T) because few iterations are required (i.e., in one iteration most/least for many intervals decreases/increases by large amounts until the minimum/maximum is reached).

```
Compute_Most_Least(intervalSize) {

  apply_constraints (I) {
    for every E_n ≥ f constraint
      N = n⌊I/f⌋ + Min(most[I mod f], n-least[f-2-(I mod f)])
      most[I] = Min( N, most[I]);
    for every E_m ≤ g constraint
      N = m⌊I/g⌋+Max(least[I mod g],m-most[g-2-(I mod g)])
      least[I] = Max( N, least[I]);     }

if (we haven't initialized)
  check for well-formed constraints (Theorem 1);
  T = Max( Max{all E_n≥f}f, Max{all E_m≤g}g);
  for I = 0 to T
    most[I] = infinity;  least[I] = 0;
  repeat
    for I = 0 to T
      apply_constraints(I)
  until (contents of arrays stop changing)

apply_constraints(intervalSize)
return most[intervalSize] and least[intervalSize]    }
```

The techniques developed above can also be used to size blocking queues. The only difference for blocking queues is that constraints on receive events have an altered semantics. If a receive event blocks, no more receive events will occur until a send event occurs. Timing constraints need to express information about the amount of time that can pass from an unblocked receive to the next receive event. Also of importance is the amount of time that can pass from a send that dispatches the blocked receive event to the next receive. These constraints can be specified within the OEgraph framework. In either type of queue, the constraints refer only to the first next receive event, and it is likely that the separation times are identical in both cases. Therefore, we use exactly the analysis presented above with a different semantics for the timing constraints on R events.

IV. Example Queue Sizing

We consider an example in which send and receive events arrive asynchronously to a non-blocking queue which is used to buffer data being transferred between processes. Send events represent either message headers or individual pieces of message data. The fastest separation time between a header or a piece of data and the next consecutive piece of message data is specified as $S_1 \geq 3$. The slowest worst case separation time between consecutive receive events is a given, $R_1 \leq 5$. Unfortunately, these worst case constraints are not sufficient to bound the size of the queue because send events can arrive on average more often than receive events.

However, there is a large delay between consecutive messages which can be specified as $S_{100} \geq 750$ (e.g., messages consist of at most 98 pieces of data and one header). This is sufficient to compute a worst-case queue size of 41 (for an interval of size 297) after considering intervals up to 501 in size. The algorithm does not consider larger intervals because at 501 the number of receives is equal to the number of sends (100) and the queue is guaranteed to have been emptied.

If it was known that receive events will be occurring more frequently than the worst case separation (e.g., interrupts in the receiving process can delay receive events but the interrupts themselves do not occur very often) then the queue size can be decreased. For example, $R_{50} \leq 200$ results in a worst-case queue size of 31 (for an interval of size 294) after considering intervals up to 401 in size.

V. Conclusion

Sizing potentially infinite queues is an important component of higher level synthesis [5]. Placing the burden on the user to set the size of such queues is an unnecessary overhead that can lead to inefficiencies in the exploitation of the inherent parallelism in the specification and detracts from the modularity and reusability of the specification.

In this paper, we have described an algorithm for sizing synchronization queues given constraints on the rate of send and receive events. These constraints are either given or can often be inferred given the presence of other constraints on the circuit's behavior (e.g., the circuit's propagation delays). In current synthesis systems, the user must explicitly specify the queue size and then ensure that it does not overflow. The algorithm presented here is a part of an incremental approach to synthesis being developed with OEgraphs as a foundation. Our immediate plans are to extend this work in the direction of synthesizing the control logic for the *physical* queue that will need to be implemented and perform tradeoffs between circuit parallelism (slowing down the rate of send events or speeding up the rate of receive events), queue size, and control logic complexity.

References

[1] M. McFarland, A. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems" Proceedings of the IEEE, Vol. 78, No. 2, February 1990.

[2] D. Ku, G. De Micheli, "HardwareC - A Language for Hardware Design Version 2.0", Technical Report CSL-TR-90-419, Stanford University, 1990.

[3] T. Amon, G. Borriello, "Operation/Event Graphs: A Design Representation for Timing Behavior" 10th International Conference on Computer Hardware Description Languages, April 1991.

[4] T. Amon, G. Borriello, "OEsim: A Simulator for Timing Behavior" 28th ACM/IEEE Design Automation Conference, June 1991.

[5] J. Huisken, et. al., "Synthesis of Synchronous Communications Hardware in a Multiprocessor Architecture" 5th International Workshop on High-Level Synthesis, March 1991.

OPERATION / EVENT GRAPHS: A Design Representation for Timing Behavior

T. Amon[a], G. Borriello[a] and C. Séquin[b]

[a]Department of Computer Science and Engineering, University of Washington, Seattle WA 98195, USA

[b]Department of EECS - Computer Science Division, University of California, Berkeley CA 94720, USA

Abstract

The specification of a digital design encompasses behavioral and structural aspects with both functional and timing components. A modern design representation must be able to capture elements in the entire space in order to support high-level synthesis, simulation, and verification tools. We propose a new model called operation/events graphs that meets these criteria. The model relies on the separation of functional and timing elements into a bipartite graph that is a hybrid of data-flow and event-graph models. Special focus has been placed on a general mechanism for timing specification using a restricted first-order predicate calculus. The model is interesting in that it has a clear semantics that make it a solid foundation for a complete set of high-level tools some of which have been implemented or are under development.

1. INTRODUCTION

Recent interest in high-level synthesis, simulation, and verification tools necessitates the development of a design representation that can capture the many facets of a digital circuit specification. These specifications consist of both behavioral and structural aspects for each of which there are functional and timing relationships to describe.

Circuit *behavior* is the high-level description of what a circuit does without overly specifying how that computation is performed. Circuit *structure* is the low-level description of how the computation is implemented, that is, the logic gates and flip-flops used. Functional aspects of both behavior and structure describe the data transformations and computations to be performed on the inputs in order to generate the outputs. In contrast, timing relationships for behavior and structure describe temporal properties such as minimum and maximum separation times for signal events. Figure 1 maps out the space of design representation schematically.

Integration of the entire space of design representation into a unified framework is critical to tool development. Incremental synthesis algorithms can be employed if a design can be evolved from behavior to structure in a single representation. A single simulator can be used to simulate the design at

Behavior	Structure	
abstract concurrent program (not necessarily implementable)	register-transfer level descriptions	Functional
real time constraints (context dependent)	propagation delays setup/hold times	Timing

Figure 1. The design representation space is divided between behavior and structure and orthogonally between functional and timing aspects. Examples in each of the four quadrants demonstrate the distinctions.

any point in the synthesis process and validate the tools and specification. Verification tools can analyze timing and functional requirements and report their satisfaction. Domain specific languages can be used to enter design descriptions which compile into the underlying representation (see Figure 2).

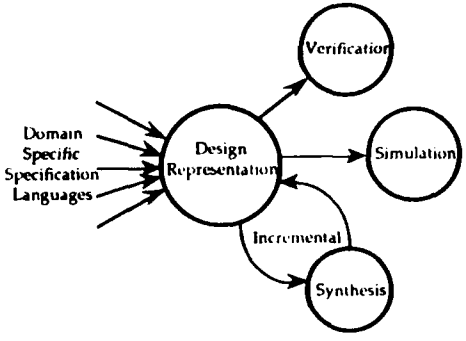


Figure 2. A unified design representation can facilitate the development of many high-level CAD tools including synthesis, simulation, and verification applications.

Existing representation methods focus on functional aspects with only rudimentary capabilities for timing information (only structural timing). Specification methods provide rich models for the transformation of data while assuming a fixed and simple timing model (usually, a single-phase synchronous clocking scheme). High-level synthesis tools employ this model to transform functional behavior into a circuit structure that performs the same data transformations in the same partial order. Simulators exist that designers can use to empirically verify that what was specified is what was desired. Verification tools can demonstrate the equivalence of behavioral and structural functionality.

Why is this state of specification, simulation, synthesis, and verification tools not adequate? These tools enable us to automatically synthesize many interesting circuits from high-level specifications. Why should a design representation also include timing behavior? The reasons are many.

Firstly, when synthesizing a circuit not all aspects of its environment are under the control of the designer. That is, the circuit must conform to the environment in which it will be placed. The environment may demand that particular timing relationships be respected. These can be as simple as setup and hold times or as complex as the spacing between messages sent over a local area network. Secondly, even within a circuit we must be able to adequately describe the timing methodologies used to implement different parts of the circuit. These include precharging constraints, pipeline interlocks, and multiple-phase clocks. Lastly, we must be able to provide an abstraction of a circuit based on its interface behavior. This is important for information hiding, that is, the ability to specify the essential aspects of a circuit's behavior while minimizing the description of its internal realization. Also, we must have the ability to describe interfaces that may not have a circuit realization at all, for example, the specification of a system backplane bus protocol and the timing constraints that must be respected for proper operation.

Recently, there has been a convergence of interest on models of timing behavior from several communities. In the traditional high-level synthesis community there has been the realization that synthesized designs must be integrated into systems that may impose timing restrictions. The speed-independent circuit design community is considering the propagation delays of internal components and the reaction time of the circuit's environment in attempts to make these designs smaller and more cost effective. There is a long standing interest in timing behavior in the real-time software systems community where problems similar to circuit synthesis, such as high-level abstraction and system integration, are increasingly important.

This paper is divided into six major sections. Section 2 describes the full range of requirements that must be satisfied for a timing specification method to be considered adequate. Section 3 outlines why existing methods do not meet these requirements. Section 4 demonstrates our approach to timing specification and provides the details of our new model. Sections 5 describes the capabilities enabled by this new timing model in high-level synthesis, simulation, and verification. Section 6 concludes the paper by summarizing the contributions.

2. REQUIREMENTS FOR TIMING SPECIFICATION

Timing relationships must be specified at many different levels for both behavioral and structural specifications. For example, a specification may include structural elements for which propagation delays are specified. It can also include Boolean equations and finite-state machines with propagation constraints and setup and hold time requirements that must be satisfied. At higher behavioral levels it must be possible to specify constraints that need only be met in a specific context (e.g., during read and not write operations) or when events have a particular causal relationship In this section, we will examine the different levels and relationships in more detail.

All physically realizable circuit elements exhibit some form of propagation delay. Output changes are delayed with respect to input changes. Actual delay values vary and are often characterized by specifying a distribution or minimum/typical/maximum values. Devices may exhibit either transport delay (transmission lines) or inertial delay (gates).

The specification for a Boolean network often contains timing requirements that an implementation of the network must meet. For example, in Figure 3, one requirement might be that a change on A propagating to B take no more than 5ns. This is an example of an input to output constraint. The specification could also contain output to output constraints (B and D change at the same time) or input to input constraints (changes in C are separated by at least 15ns.) which are not constraints on the implementation but instead provide information about the circuit's environment. Such information may result in a better implementation (for example, if a race condition can be ignored).

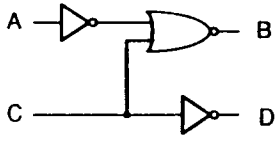


Figure 3. A Boolean network that demonstrates input to output as well as input to input and output to output timing constraints.

In a sequential network, clocking methodologies are an essential part of the circuit specification and any conceivable methodology should be specifiable.

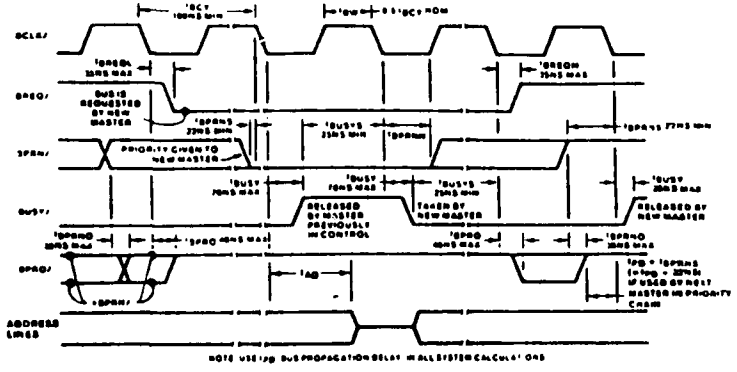


Figure 4. A timing diagram from the specification of the Intel Multibus [1]. Constraints specify separation times between *events* (i.e., changes in logic level on signal wires).

This includes designs that are: asynchronous, synchronous (single-phase or multi-phase), edge-triggered, level-sensitive, etc. Typically, timing constraints specify: clock rates, setup times, hold times, and separation times — both absolute ("5 ns") and relative to a clock ("3 cycles"). These constraints are often specified using timing diagrams and tables. Figure 4 shows an example of such constraints from the specification of the Intel Multibus [1].

In behavioral specifications, timing constraints can be much more abstract. Requirements specify separation times between abstract behavioral events. Constraints may apply only in a specific behavioral context or as a result of a specific causal path. For example, in Figure 5, requests to a non-FIFO queue need to be acknowledged in less than 100ms. The constraint could also be conditional, depending upon who issued the request (i.e., low priority requests need to be acknowledged in 1000ms.).

Other examples include the specification of constraints between events synchronizing the execution of concurrent loops, and specifying the average rates at which inputs arrive at an interface. In fact, interface timing requirements are an important example of high-level timing constraints [11].

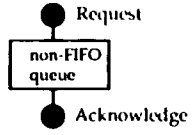


Figure 5. A non-FIFO queue that acknowledges requests in a different order than the requests were received.

3. EXISTING APPROACHES TO TIMING SPECIFICATION

No existing approaches to timing specification meet all of the requirements described in the previous section. Some models do capture some levels of timing specification but do not span all the levels. Others simplify the problem by only considering a subset of the possible timing behaviors. There are aspects, especially at the higher-levels of specification, that are not considered by any existing approach. In this section, we will describe the different classes of timing specification methods and their deficiencies.

3.1 Hardware Description Languages

Hardware description languages characteristically include very little support for the specification of timing requirements. Most are based on a synchronous model that limits the circuits that can be described. Timing properties have been introduced in an ad hoc manner and are not a first class aspect of the specification [2]. HDLs are typically used to generate data-flow graphs which have formed the basis of most high-level synthesis work [3,4,5]. These languages are useful in that they usually provide a simulatable specification. However, when simulation efficiency becomes an overriding concern and leads to language features specifically for simulation, synthesis tools suffer from the inability to synthesize these simulation constructs. This

has been the case for VHDL and has led to subsetting and policy of use efforts that dilute the standardization efforts and further confuse synthesis tool developers. Supersetting efforts have also attempted to add new language features to VHDL specifically for synthesis, leading to two disjoint views of the circuit: one for simulation and one for synthesis [6]. Of course, it is then necessary to verify that these two views are compatible, a potentially intractable problem.

3.2 *Petri nets and other graph models*

Petri nets have been used for a wide range of formal specification applications and have been extended in several ways to include constructs for describing simple delays (e.g., timed Petri nets) [7,8]. The semantics of Petri nets that make them powerful for behavioral specification (i.e., token propagation) make them difficult to use for describing hardware directly. Specifically, the non-persistency of enabled transitions is difficult to embody in hardware. Event graphs have been developed for the specification and synthesis of asynchronous circuits that enforce persistency [9,10]. However, data transformation and complex looping and conditional control structures are not included. One of the attempts to perform this integration has led to the work described in this paper [1, 2]. Another comprehensive graph model is the Design Data Structure [13,14]. It uses multiple independent hierarchies for timing, functional, and structural aspects of the specification making it difficult to simulate directly and to manipulate because of the many bindings between the three graphs.

3.3 *Temporal Logics*

Temporal logics are often imbedded within other representations which rely upon the logic to describe the temporal properties of the specified system [15,16,17]. They are capable of describing many complex timing relationships. One critical deficiency, however, is their inability to describe causal relationships. For example, in the non-FIFO queue of Figure 5, the chronological order of request and acknowledge events cannot be used to infer which request/acknowledge pairs are to be constrained. This deficiency has never (to our knowledge) been addressed in the literature and temporal logics are often regarded as being complete with respect to the expression of timing properties. Causality is also not expressible in other formal representations such as waveform algebra [18].

3.4 *Automata Models*

Automata models have recently gained much popularity in describing communicating state machines and the constraints on their behavior. Automata can be described to check for proper input and output sequences and can be extended to include timing constraints [19,20]. The difficulty in finite automata models lies in their inability to handle potentially infinite sequences (e.g., counting), that is, they cannot keep track of an unspecified number of events without an explosion of states. Furthermore, like the Petri net and temporal logic models, they are inadequate for describing circuit structures as simple as buses.

4. A NEW MODEL FOR TIMING SPECIFICATION

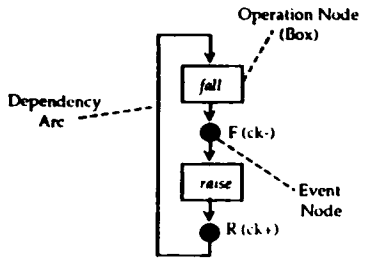
The representation we are proposing is a straightforward graph model whose basis consists of two types of nodes connected by a directed arc to form a bipartite graph. Both types may be hierarchical.

4.1 *Basic Elements*

Event nodes represent changes in logic level on circuit wires. If the event node is on a cycle in the graph then it may occur multiple times. Therefore, it is important to distinguish a *discrete* event from an event node. An event node represents the event that will occur, a discrete event is an actual occurrence of that event. The logic transition of a discrete event is to any member of the enumerated set of values that may be carried on the signal wire on which the event takes place (e.g., high, low, don't care, tri-state, undefined, etc.). The transition can be attached to an event either statically (i.e., the event always represents a high transition) or dynamically (i.e., when the event is generated). Event nodes are used to express timing properties of behavior and structure.

Operation nodes or *boxes* are the second type of node in our graph representation and correspond to the functional aspects of behavior and structure. Boxes contain program code that is evaluated whenever an input event occurs (e.g., C++ source code). The evaluation may conditionally generate output events which will occur at some future point in time and/or possibly change internal state. An example of an operation node is a logic gate that may generate an output event whenever an input event occurs. The delay in generating the output event corresponds to the propagation delay of the gate (the delay is specified by either a fixed value or a range of values and a distribution function to indicate uncertainty with respect to when the event will actually occur). A more abstract example of a box is one that forks two independent processes: an input event arrives at an operation node that will then cause two parallel output events thereby permitting two parts of the specification to proceed in parallel. The events, in this case, do not correspond to logic transitions and instead represent abstract control flow.

Dependency arcs specify the flow of events in to and out of boxes. The graph is bipartite because dependency arcs specify flow of control by directing events into boxes and the output of boxes to events. We purposely make no distinction between data and control to permit optimization and synthesis algorithms full flexibility in making or not making the distinction [3]. Events have an in-degree of at most one dependency arc, thus each event is either an external event or is caused by the execution of a single operation node. Events may have arbitrary out-degree.



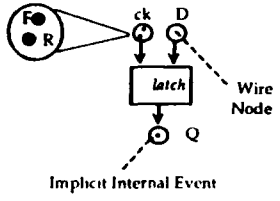

```
oe_wire ck("ck");
oe_event F("F",ck,LOW);
oe_event R("R",ck,HIGH);

fall() { if (trigger==R) cause(F, 25);}
raise() { if (trigger==F) cause(R, 25);}

main {
 oe_box op_fall("op_fall", fall);
 oe_box op_raise("op_raise", raise);
 connect(op_fall,F);
 connect(R,op_fall);
 connect(F,op_raise);
 connect(op_raise,R); }
```

Figure 6. Graphical and textual versions of a simple single phase clock in our representation (from a behavioral perspective).

Event nodes that affect changes on the same wire can be grouped into an event node set. Since every possible change on the wire (every discrete event) is collected into the set we refer to such a set as a wire. In practice, wires can be considered a third type of node in our graph. Operations may have wires as inputs (any event that is a member of the wire is an input to the operation) and may also have wires as outputs (the output wire's value is changed via an implicit internal event). An operation can ask about the *value* of an input wire (i.e., the *effect* of the most recent discrete event on that wire).




```
oe_wire ck("ck");
oe_wire D("D");
oe_wire Q("Q");

latch() {
if (trigger==ck && ck==HIGH)
 cause(Q, uniform_delay(5,10), valueOf(D));}

main {
 oe_box op_latch("op_latch", latch);
 connect(ck,op_latch);
 connect(D,op_latch);
 connect(op_latch,Q);}
```

Figure 7. A clocked edge triggered D latch in our representation. Note that the propagation delay is uniformly distributed between 5 and 10 time units.

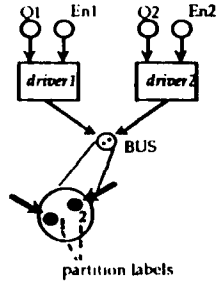
Operations having wire inputs and outputs represent functionality from a structural perspective. Operations having event inputs and outputs represent functionality from a behavioral perspective. Mixed perspectives are also possible (e.g., an operation having wire inputs carrying data and event inputs signalling flow of control).

In most cases, the effect of an event on a wire corresponds exactly to the transition that the event represents. However, for a bus this is not necessarily the case. With two or more operations driving a bus, an event asserting a tri-state value may leave the bus connected to ground (and not disconnected) because there is another operation connected to the bus. Normally, any wire

that has an indegree of more than one is a bus (i.e., two or more operations are driving the wire). Two or more events on the same wire may or may not represent a bus, depending upon whether or not synthesis transforms the events into wire outputs of one operation or two. This transformation is guided through the use of *partition labels* which can be attached to events.

Partition labels are specified when an operation is connected to an output event or an output wire (to label the operation's internal implicit event for the wire). Two events with different partitions thus represent a behavioral perspective of two events interacting on a bus. If partition labels are not specified events default to the same partition.

Every wire also has a physical model which is used to determine how events affect wires. For example, if a wire (bus) is connected to both ground and power by different operations, the wire's value is undefined assuming a direct-connect model. Other interesting physical models have also been defined (e.g., precharged wires).




```
oe_wire En1("En1"), En2("En2");
oe_wire Q1("Q1"), Q2("Q2"), BUS("BUS");;

driver1() {
if (En1==HIGH) cause(BUS, 10, valueOf(Q1);
else cause(BUS, 10, TRI);}
driver2() {
if (En2==HIGH) cause(BUS, 10, valueOf(Q2);
else cause(BUS, 10, TRI);}

main {
 oe_box drv1("drv1", driver1);
 oe_box drv2("drv2", driver2);
 connect(Q1,drv1); connect(Q2,drv2);
 connect(En1,drv1); connect(En2,drv2);
 connect(drv1,BUS,1); connect(drv2,BUS,2);}
```

Figure 8. A bus being driven by two operations. If both enables were high and Q1≠Q2 the BUS would be driven undefined. This behavior relies on the physical model used for the bus and the separate partitioning of each operation's internal implicit event on the bus.

4.2 Relationships Between Events

Although simple timing constraints have an obvious representation (i.e., a labelled directed edge in the graph representing the difference in time between two events) the representation of constraints between events nested within loops, forks, conditional branches, and concurrent structures requires a more comprehensive mechanism. The problem is that constraints are relationships between discrete events — not event nodes. For example, consider a graph with two events: F and R which represent the falling and rising edges of a clock (see Figure 6). To describe the rise to fall time as a constraint from R to F, is incorrect. Rather, the constraint applies to pairs of discrete occurrences of R and F and we can rely on chronological relationships to identify the discrete events. A timing constraint exists from every discrete R event to the next chronologically occurring discrete F event. Identifying discrete events is problematical in the general case. Chronological relationships can often be

used to identify the discrete events involved. However, constraints may be relative to a particular execution path in a complex graph, and chronological relationships alone are not sufficient. Constraint specification must include a way of getting at this history.

We have developed the concept of *event ancestry* to address this issue. Event ancestry is specified by naming the events and internal state that are used to generate an output event or internal state change. Every discrete event has an ancestry tree, consisting of its immediate ancestors and their ancestors, transitively. An ancestor of a discrete event is any previously occurring discrete event that led to the generation of its descendant through its effect on a series of boxes. Whenever an operation decides to generate an event, the new event has as ancestors the most recent discrete occurrences of the input events named as ancestors. Intuitively, the ancestors of an output event are the input events that were used to determine whether to generate the output event. When an input wire is named as an ancestor, the most recent discrete event on the wire (as seen by the operation) is the ancestor. When a state variable is named as an ancestor, the discrete event ancestors of the state variable are all ancestors. Whenever an operation decides to change its internal state, the state variable's ancestry changes in an analogous manner. This is necessary if we are going to be able to decompose the internals of a box into more primitive boxes.

4.3 Timing Constraints

The language used for the specification of timing constraints (which are assertions about the desired behavior of the specification) is based on a first-order predicate calculus that consists of:

- standard Boolean relations ($\vee,\wedge,\rightarrow,\sim$).
- existential and universal quantification of discrete events ($\exists,\forall$).
- arithmetic functions and relations for integers ($+,-,<,>,=$).
- an integer function "τ" that returns the time at which a discrete event occurs.
- a relation "ancestor" to test whether a discrete event is an ancestor of another.
- a function "v" returning the wire value for a discrete event.
- other wire value functions and relations (and, or, invert, $=$, $\supseteq$, etc.).

The time at which an event occurs is not an infinite precision real, but instead is an integer. Thus multiple events may appear to occur at the same time, due to a natural granularity problem that also exists in the physical world (i.e., at some level of detail it is not possible to determine which of two events actually occurred first). This affects the definitions of the primitive relations described later in this section.

The full first-order predicate calculus introduces problems which we have addressed by restricting the representation of timing constraints to the following format: (1) universal quantification of the discrete events involved in the constraint, (2) specification of the context within which the constraint must hold and, (3) specification of a particular timing relationship that is required to be true when the context is true.

quantify discrete events + context $\Rightarrow$ timing requirement for the discrete events

These simplifications were made for a number of reasons. First, event generation (existence) is represented in the functional components of the graph. If an event is required to exist given a particular set of circumstances, an operation can be defined which generates the event given the circumstances. Therefore, existential quantification — which is used primarily to constrain functionality, is already encapsulated within our representation. Furthermore, if such a constraint was not always satisfied, it would be very unclear how synthesis could transform the graph so that it was.

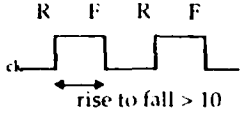
In simulation, the universe of discrete events changes as new events occur and are added to the universe. Constraints are statements about the infinite universe of events. If the universe is constantly changing, when can constraints be checked and violations reported? Violations can be detected and reported for constraints with universal quantification because a particular instantiation of discrete events causes constraint violation. With existential quantification violations often can never be reported because the events that satisfy the constraint may not yet have occurred. Of course, we can report satisfaction for existentially quantified constraints and not for universal ones, but in simulation, constraint violations are of primary interest. Often we would not be able to conclude anything about constraints that contain both quantifications and the simulator would be extremely inefficient because constraints would be repeatedly rechecked.

In order to capture much of the expressibility of the full calculus, while restricting it to the format described above, we added the following four relations because they encapsulate concepts which are essential for timing constraint expression (lower case variables represent quantified discrete events):

- **mra(x,S,y)** $\equiv \forall z$ (ancestor(x,y) $\wedge$ ($z \notin S \vee \sim$ancestor(x,z) $\vee \tau(z) \leq \tau(y)$)), to test whether x's most recent S ancestor (of all the events in the set S) is/might be y.
- **pco(x,Y,y)** $\equiv \forall z$ ($y \neq x \wedge \tau(y) \leq \tau(x) \wedge (z \notin Y \vee \tau(z) \geq \tau(x) \vee \tau(z) \leq \tau(y))$), to test whether x's previous chronological occurrence of a Y is/might be y.
- **nco(x,Y,y)** $\equiv \forall z$ ($y \neq x \wedge \tau(y) \geq \tau(x) \wedge (z \notin Y \vee \tau(z) \leq \tau(x) \vee \tau(z) \geq \tau(y))$), to test whether x's next chronological occurrence of a Y is/might be y.
- **changes(x,W)** $\equiv \exists w \, \forall z$ ($(w \in W \wedge (v(x)=dc \vee v(x)=valid \vee v(x) \neq v(w)) \wedge w \neq x \wedge \tau(w) \leq \tau(x) \wedge (z \notin W \vee \tau(z) \geq \tau(x) \vee \tau(z) \leq \tau(w))$), to test whether x may have changed the value of its wire W.

They have semantics that are easily implemented by the simulator and the four relations do not introduce complications which would prevent efficient incremental constraint checking — an important additional benefit of our restrictions. Of course, restricting the calculus weakens its expressibility. However, based on experience with many example specifications we believe that there is no effect with respect to the specification of timing requirements encountered in digital systems.

Figure 9 contains the specification for the rise to fall time constraint discussed in section 4.2. The textual representation for the calculus is very straightforward. For every event and wire there is a set (having the same name) which represents all of the event's or wire's discrete events. Discrete events are quantified and constraints are specified as described above. Constraints can be specified individually as shown in Figure 9 or through the use of subroutines that can be parameterized (as is the case with common constraints such as setup and hold). Further details can be found in [21].



inserted into main() of figure 6

```
oe_constraint rise_to_fall("rise_to_fall");
discrete_name F0("F0",F);
discrete_name R0("R0",R);
rise_to_fall.quantify(F0,R0);
rise_to_fall context(nco(R0,F,F0));
rise_to_fall require(timeOf(F0)-timeOf(R0) > 10);
```

constrained calculus: $\forall f \forall r ((f \in F \wedge r \in R \wedge nco(r,F,f)) \rightarrow (\tau(f) - \tau(r) > 10))$
full calculus: $\forall f \forall r \exists z ((\tau(f) - \tau(r) > 10) \vee (f \notin F \vee r \notin R \vee \tau(f) < \tau(r) \vee (z \in F \wedge \tau(r) < \tau(z) < \tau(f))))$

Figure 9. Our representation for the rise to fall time constraint described in 4.2

4.4 Examples

Section 2 described various levels and types of timing requirements. Section 4.1 described how propagation delays are specified. Our representation specifies inertial and transport delays as well as rise/fall times using two constructs: wire durations (e.g., rise and fall times) and input sensitivity levels (e.g., 10%, 50%, or 90%), for more details see [21].

In the following examples quantified discrete events are identified by specifying the name of the event node subscripted with an integer. We omit the detailed textual representations found in Figures 6-9 and instead rely on a graphical representation. Figure 10 contains examples of the input to output and input to input constraints on the Boolean network described in Figure 3 of section 2. Constraints are drawn as dashed lines from the wire nodes quantified by the constraint.

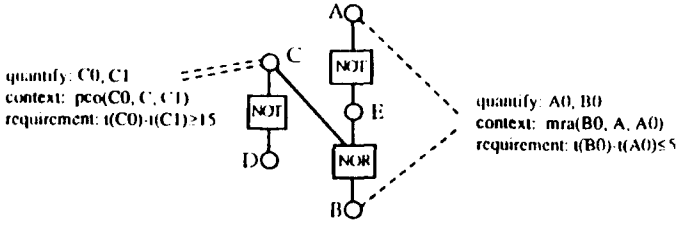


Figure 10. Combinational logic with input to output and input to input constraints.

Figure 11 contains a constraint that states that events A and B are required to occur exactly one cycle apart whenever A and B occur from the same request event (REQ). The constraint is complicated by the fact that A and B occur synchronously to the falling edge of the clock (F) and may occur anywhere within the shaded region shown in figure 10 (requiring an exact separation time between A and B would thus be incorrect). This is an example of the type of constraint that cannot be expressed without the use of event ancestry. For example, waveform algebra [18] cannot be used to describe this constraint because it cannot pair up the A and B events.

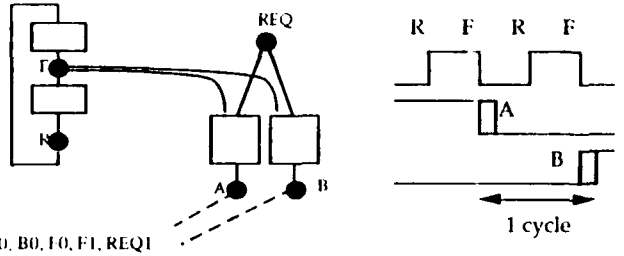


Figure 11. A sequential logic constraint requiring two events (A and B) to be one cycle apart.

The context for the constraint establishes the identities of the quantified events. F0 is the clock edge prior to A0; F1 is the clock edge prior to B0; A0 and B0 share the same REQ ancestor. The requirement is simply that F0 be the clock edge prior to F1 or that F1 be the clock edge prior to F0. Figure 12 shows how the Req and Ack events of the non-FIFO queue from Figure 4 are easily paired up using event ancestry.

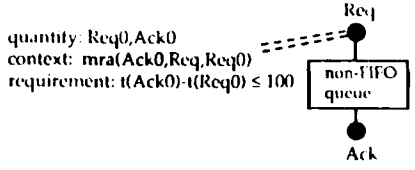


Figure 12. A behavioral constraint that requires the notion of event ancestry.

Figure 13 shows the OEgraph for the Intel Multibus [1]. The graph represents the master read and master write operations including the arbitration sequence for the bus and the data transaction itself. Timing constraints are not shown in Figure 13 although there are over 40 in the specification.

The Multibus example highlights many of the features of OEgraphs. The specification includes both synchronous and asynchronous components. One

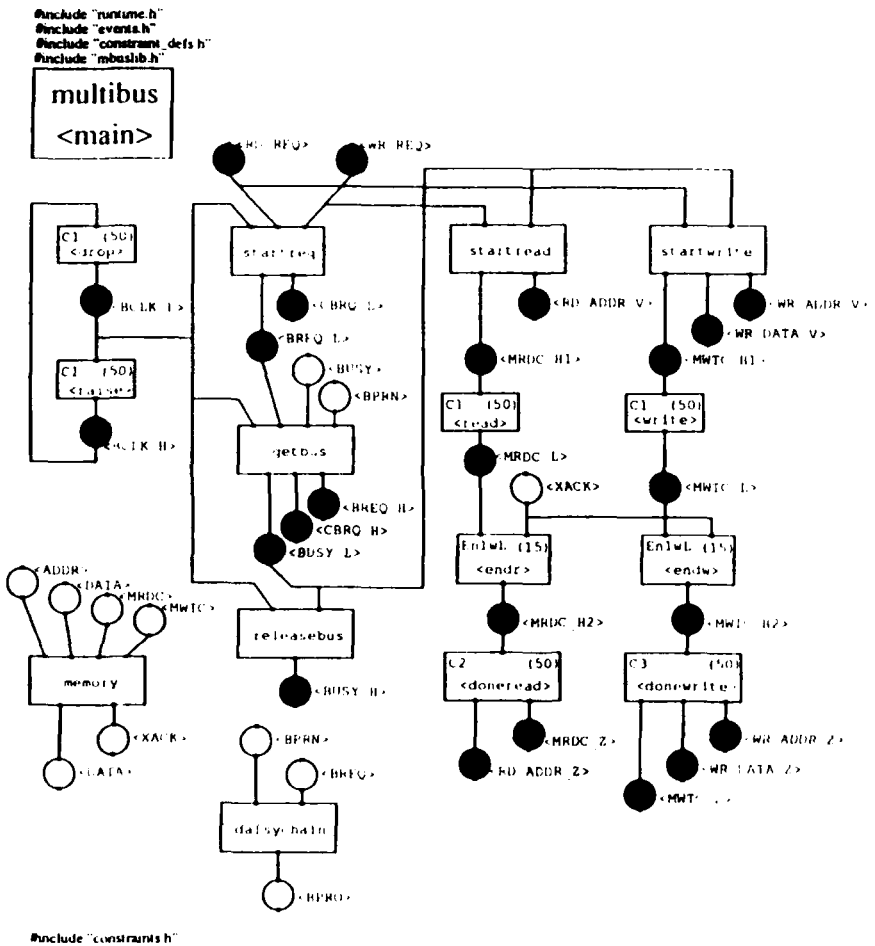


Figure 13. Specification of the Intel Multibus master read and write operations

can see that clock events are only used by the subset of operation nodes that deal with arbitration. The data transaction proceeds asynchronously to the system clock. Behavior and structure are both represented. The signalling conventions used for arbitration and the handshaking used for the data transaction are represented primarily with event nodes. The daisy-chain for arbitration resolution and the memory module responding to the requests are represented with wire nodes and are structural entities. The daisy-chain operation node is a simple NOR gate. The memory module is part of the environment of the Multibus as are the read and write request events at the top of Figure 13.

An important thing to note about this example is that it was derived directly from the Multibus specification. Three timing diagrams were translated into the sequences of events representing the arbitration, data read, and data write operations. The daisy-chain logic was obtained from a logic diagram in the specification. The timing constraints were derived from the table of timing requirements in the Multibus specification. The advantage of OEgraphs is that all these disparate forms can be combined into a single internal representation with a formal semantics for each construct.

5. TOOLS

The operation/event graph (*OEgraph*) facilitates the development of a complete set of high-level tools. In Figure 2, four general categories where represented schematically: compilers, synthesizers, simulators, and verifiers. Each of these can benefit from the generality of the model and can be linked together to provide a complete design synthesis environment.

5.1 Compilers

OEgraphs are very expressive. Structure and behavior can be described in a unified framework and functional as well as temporal requirements can be specified. Our representation supports specifications that are either textual (see Figures 6-9) or graphical (the Multibus specification in Figure 13 was created using a graphical editor which compiles equivalent text). We envision using higher level tools to specify different components of the specification.

For example, behavioral specifications can be entered using a concurrent programming language. The statements of the language are translated into operation nodes. Data flow including synchronization primitives such as send/receive can be encapsulated in events routed to the appropriate operation nodes much in the same way as traditional data flow graphs. Control flow can be represented by abstract events linking the operation boxes in a maximally concurrent partial order. Timing constraints can be represented by linking timing declarations to statements in the language or through the use of timing diagram editors [2, 11]. Such constraints are easily translated into the restricted calculus. Partial structure as specified by a schematic editor, including propagation delay information, can also be included in the description.

More specialized input languages, including graphical editors, may be appropriate for digital signal processing, graphics computations, etc. Each domain may choose its own medium and compile to the OEgraph model.

5.2 Synthesizers

Synthesis, in the context of OEgraphs, is the transformation of the graph so that behavioral operations and event nodes are turned into structural operations and wires. This entails deciding how each operation is to be implemented in terms of sequential logic primitives. Operations that generate events that affect the same wire must be grouped into a single operation that controls that wire. Alternatively, a bus structure, where two operations drive the same wire, may be more appropriate. Operations that have events as inputs must be modified to distinguish the event on a wire (differentiate between that particular event and all other events on that wire). This may entail adding other input wires to the box that can be used to disambiguate the events.

Besides accomplishing this mapping of behavior to structure, synthesis must also ensure that timing constraints are satisfied. For each operation node in the graph, allowable propagation delay values must be determined so that they can be properly mapped to logic primitives. A tight constraint may necessitate a larger and more parallel circuit than would otherwise be warranted.

More interestingly, synthesis algorithms must also create structure previously represented implicitly in the graph. A simple example of this is the placement of synchronizing elements between asynchronous and synchronous portions of a design. Another example involves sizing a queue. A structure to buffer the data (and its requisite control signals and events) may need to be generated. In the original behavioral specification, there was no reason to specify a size for the queue , it should be as large as necessary. As operations are turned into structure with more realistic propagation delays, the size for the queue must be determined by the synthesis algorithm using the component delay and timing properties of the input and output data streams. In some cases, user specified behavior may not be implementable, and synthesis should inform the user that more refinement or additional constraints are needed.

Our approach leads to a more modular view of synthesis that separates the functionality of a circuit's internals from its interface signalling conventions and timing constraints [12]. The same functional description of a circuit's internals can be used for different environments. In other words, it should not be necessary to change the specification of a computation just because we are moving the circuit from one bus protocol to another. Synthesis algorithms must determine the appropriate internal structures to match the environment. For example, in one case a queue will be needed and in another it may not. This is to be contrasted with current high-level synthesis systems where the specification must be rewritten because the specification of the internals and the interface are intertwined.

5.3 Simulators

It is of critical importance that a design representation can be validated by the user. Making sure that what was specified is what was desired is the first step in verifying a design and cannot be automated. A simulator provides the user with the capability to try out the circuit and make sure it behaves as expected.

OEgraphs were designed with simulation in mind. A clear simulation semantics was a requirement for all the features of the model. The most important goals included modeling of real hardware and general timing constraints. Modeling of real hardware includes bus structures, propagation delays (both inertial and transport), and events with rise/fall times. These features are all included. The difficulty with timing constraints was ensuring that the calculus used for their specification had a clear semantics in the simulation world and would support incremental constraint checking. This was accomplished with the four new primitives and format restrictions outlined in section 4.

The OEsim simulator has been completed and is in use by colleagues at other universities. It has already proven instrumental in debugging specifications. OEsim is a compiled simulator that takes as input a C++ description of an OEgraph and compiles it linked to the simulator front-end. OEsim provides incremental constraint checking and performs many optimizations for simulation efficiency. These include deciding when constraints should be checked and garbage collection of old events that are no longer needed for constraint checking. By virtue of being a compiled C++ program, an operation can include arbitrary C++ code that can be used to provide special interactions with the user and read and write data files.

Simulation provides empirical verification that for a given set of input vectors no constraints are violated. OEsim could be extended to support more comprehensive testing/verification methods (e.g. worrying about coverage). More complete verification tools are clearly needed.

5.4 Verifiers

An advantage of the OEgraph model in verification is that the same representation is used for all stages of synthesis. Therefore a formal verification tool need only deal with one input format. A useful verification tool would read a graph description and determine if all the constraints are satisfied. Alternatively, it could specify the allowable range of delays possible in operation nodes that would satisfy the constraints. Timing constraints are transformed as the specification moves from behavior to structure so that they can still be checked during simulation and be formally verified. This provides a check on each step of the synthesis algorithm with respect to timing.

One approach is to translate operation node programs into a first-order predicate calculus as well. This would then enable formal proof techniques to demonstrate constraint satisfaction. Also, safety and liveness conditions could be checked as well. Theorem provers which rely on human assistance [22] may be needed for some constraints and/or circuits for which analytical methods prove insufficient. An alternative would be to translate a specification into a timed automata model [19,23]. However, this would restrict verification to

specifications that do not include potentially infinite structures (i.e., arbitrary counts).

The OEgraph model is quite general allowing arbitrary C++ code in operation nodes. This generality makes the formal verification problem virtually impossible. Clearly, a policy of use must be defined that lends itself to verification. Any such policy will restrict the classes of specifications that can be verified, although such restricted classes should still be quite interesting and useful. This belief has recently been supported by [24] which reports formal verification techniques for a representation containing simplified concepts of events and timing constraints.

6. CONCLUSION

We have outlined a new unified behavioral/structural representation for digital circuits. It gives time and timing constraints their long deserved equivalent status with functional specifications. We feel that it will provide a target for the development of domain-specific description languages and a basis for incremental synthesis algorithms. This approach will eventually provide the user more control and confidence over the synthesis process by permitting simulation and verification of the design at various stages of realization including initial specification and final implementation. It will also make possible the integration, comparison, and verification of a wide variety of tools built on top of a common representation.

The definition of our graph representation is complete. We have used our simulator to describe a wide range of examples derived from real circuits or extracted from the specification and synthesis literature. We are distributing it to other researchers and it has proven itself quite useful in debugging the specifications of a collection of designs.

We are currently working on synthesis algorithms that tackle the problem of determining the implicit structures required when an abstract description of a computation needs to be connected to a real world interface. Attention is also being given to verifying that component delays will lead to a circuit that meets all its timing constraints and determining the allowable value of those delays when they are not specified. The simulator has been completed and is in limited distribution. Finally, we are looking into input languages for data communication circuits and interface specifications.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge numerous discussions with Richard Newton and David Dill. Wayne Winder and Karen Bartlett were instrumental in the implementation of the simulator. Jane Sun at UC Berkeley has been daring enough to put our simulator to the test.

This work was supported by the National Science Foundation under a Presidential Young Investigator Award (MIP-8858782) and by the Defense Advance Research Projects Agency under contracts #N00014-88-K-0453 and #N00039-87-C-0182.

REFERENCES

[1] *Intel Multibus Specification*, Intel Corporation , 1982.

[2] J. Nestor, "Specification and Synthesis of Digital Systems with Interfaces," Technical Report CMUCAD-87-10, Department of Electrical and Computer Engineering, Carnegie-Mellon University, April 1987.

[3] M. Barbacci, "Instruction Set Processor Specification (ISPS): The Notation and Its Applications," *IEEE Transactions on Computers*, January 1981.

[4] M. McFarland, "The VT: A Database for Automated Digital Design," Technical Report DRC-01-4-80, Design Research Center, Carnegie-Mellon University, December 1978.

[5] M. Shahdad et. al., "VHSIC Hardware Description Language," *IEEE Computer, Vol. 18, No. 2* , February 1985.

[6] L. Augustin, B. Gennart, Y. Huh, D. Luckham, and A. Stanculescu, "An Overview of VAL," Technical Report CSL-TR-88-367, Stanford University, 1988.

[7] J. Coolahan, Jr and N. Roussopoulos, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Transactions on Software Engineering, Vol. SE-9, No. 5* , September 1983.

[8] C. Petri, "Fundamentals of a Theory of Asynchronous Information Flow," *Proceedings of the IFIP Congress 1962, Munich, North-Holland*, 1962.

[9] T. Chu, "On the Models for Designing VLSI Asynchronous Digital Systems," *Integration, The VLSI Journal, Vol. 4*, August 1986.

[10] T. Meng, R. Brodersen, and D. Messerschmitt, "Asynchronous Logic Synthesis for Signal Processing from High Level Specifications," *Proceedings of the International Conference on Computer Aided Design*, November 1987.

[11] G. Borriello, *A New Interface Specification Methodology and its Application to Transducer Synthesis*, Ph.D. dissertation, University of California at Berkeley, 1988.

[12] G. Borriello, "Combining Event and Data-Flow Graphs in Behavioral Synthesis," *Proceedings of the International Conference on Computer Aided Design*, November 1988.

[13] S. Hayati and A. Parker, "Automatic Production of Controller Specifications from Control and Timing Behavioral Descriptions," *Proceedings of the 26th Design Automation Conference*, June 1989.

[14] D. Knapp and A. Parker, "A Data Structure for VLSI Synthesis and Verification," Technical Report CRI-85-19, Department of Electrical Engineering, University of Southern California, August 1985.

[15] E. Clark, E. Emerson, and A. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2* , April 1986.

[16] D. Harel, "Statecharts: Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 1987.

[17] A. Pneuli, "The Temporal Logic of Programs," *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[18] L. Augustin, "Techniques for High-Level Synthesis and Specification: Waveform Algebra," *Fourth International Workshop on High Level Synthesis*, October 1989.

[19] R. Alur and D. Dill, "Automata for Modelling Real-Time Systems," *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[20] R.P. Kurshan, *private communication*, 1990.

[21] T. Amon and G. Borriello, "OEgraphs and OEsim," Technical Report TR90-01-17, Department of Computer Science and Engineering, University of Washington, January 1990.

[22] M. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware," In *Formal Aspects of VLSI Design*. North-Holland, G. Milne and P.A. Subrahmanyam (Editors), 1986.

[23] A.A. Bestavros, "The Input Output Timed Automaton: A Model for Real-Time Parallel Computation," *First International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, August 1990.

[24] A. Martello, S. Levitan, and D. Chiarulli, "Timing Verification Using HDTV," *Proceedings of the 27th Design Automation Conference*, June 1990.